Protocol Specification and Analysis in Maude*

G. Denker† and J. Meseguer
Computer Science Laboratory, SRI International, Menlo Park, CA 94025, USA
{denker,meseguer}@csl.sri.com

C. Talcott
Computer Science Department, Stanford University, Stanford, CA 94305, USA
clt@sail.stanford.edu

Abstract

This paper proposes rewriting logic as an executable specification formalism for security protocols that offers some novel advantages. A message-passing object-oriented approach seems particularly natural for communication protocols and can be naturally formalized in rewriting logic. This is illustrated by using the Needham-Schroeder Public-Key protocol as a running example. The rewriting logic-based Maude interpreter [CELM96] offers also some useful advantages. Efficient executability allows prototyping and debugging of protocol specifications. But since a concurrent system can have many different behaviors, to properly analyze the system it becomes important to explore not just the single execution provided by some default strategy, but many other executions. Maude supports user-defined execution strategies, including strategies such as breadth-first-search that can exhaustively explore all the executions of a system. This is very helpful in uncovering security flaws under unforeseen attack scenarios such as those found for NSPK. We also discuss future developments along of this work, including (1) narrowing using symbolic execution techniques, (2) modularity and compositionality issues in formal reasoning, and (3) combination of rewriting logic and temporal logic.

1 Introduction

This paper proposes rewriting logic [Mes98] as an executable specification formalism for security protocols that offers some novel advantages. The level of specification is detailed enough to capture essential operational aspects and to

*Supported by DARPA through Rome Laboratories Contract F30602-97-C-0312, by Office of Naval Research Contracts N00014-95-C-0225 and N00014-96-C-0114, and by National Science Foundation Grant CCR-9633363.

†Work reported here was supported under HSPIII by German Federal and State Government through DAAD.

allow rapid prototyping and debugging by directly executing the specifications. In this sense, it can be compared to protocol specifications based on process algebras, CSP, the π-calculus, UNITY, or Petri nets. However, unlike those approaches, rewriting logic does not prescribe a fixed model of concurrency but is instead a general semantic framework in which many different concurrency models, or hybrid combinations of those models, can be naturally expressed [Mes92, Mes96b]. This translates into additional naturalness and ease of representation. For example, a message-passing object-oriented approach in the style of the Actor model seems particularly natural for communication protocols, but does not come naturally when a synchronous model is adopted. In Section 2 we explain how rewriting logic and the rewriting logic-based Maude language [CELM96] can naturally formalize concurrent object systems which may be either asynchronous or synchronous, and in Sections 3–4 we illustrate the naturalness of an object-oriented protocol specification in Maude using the Needham-Schroeder Public-Key (NSPK) protocol as a running example.

The Maude interpreter [CELM96] offers also some useful advantages. The first is efficient executability, that can be traced in detail to follow each execution step; this has allowed us to prototype and debug the NSPK specification, making explicit in the process some subtle points usually left implicit in a typical textbook-style specification. This suggests the possibility of automating the passage from textbook specifications to formal specifications in Maude, making explicit important implicit knowledge in the process, very much in the spirit of Jonathan Millen's CAPSL language project [Mil97]. But executability is actually not enough. Since a concurrent system can have many different behaviors, to properly analyze the system it becomes important to explore not just the single execution provided by some default strategy, but many other executions. The reflective capabilities of rewriting logic and Maude [CM96] are very helpful in this regard, because they allow user-defined execution strategies that can be formally specified by rewrite rules at the metalevel, including strategies such as breadth-first-search that can exhaustively explore all the executions of a system. This is very helpful in uncovering security flaws under unforeseen attack scenarios such as those found for NSPK. In Section 5 we present the breadth-first-search strategy we used for analyzing the NSPK protocol.

Our systematic search for an attack using a breadth-first strategy illustrates a form of model checking in which we explore all the possible behaviors beginning with a given state using the reflective capabilities of Maude. In Section 6.1 we propose the technique of narrowing to explore behaviors which start no just from one state, but from a, possibly infinite, set of states. Another important issue discussed in Section 6.2 is modularity of protocols. It is often important to harden a system against a variety of threats. If we have protocols protecting against several such threats, how can we modularly combine them to protect the system against a combined attack? Our main point is that formal reasoning about such modular combinations can be carried out at the level of their rewriting logic specifications. To achieve such modular compositions we propose using actor reflection techniques whose semantics can also be specified in rewriting logic. Finally, Section 6.3 discusses the important issue of combining executable specifications in Maude with more abstract specifications of system properties in temporal logic. The semantic nexus between the two levels of specification is given by the initial model of the rewrite theory specifying the system, which plays at the same time the role of the Kripke structure about which specific temporal logic properties are asserted.

2 Object-Oriented Specification in Maude

Maude [CELM96] is a multi-paradigm executable specification language based on rewriting logic [Mes92]. Maude integrates an equational style of functional specification in its equational logic sublanguage with an object-oriented specification style for object systems that can be highly concurrent and nondeterministic. Maude modules are rewrite theories whose basic axioms are rewrite rules. A rewrite theory axiomatizes a concurrent system, with each rewrite rule corresponding to a firing pattern for each of its basic concurrent transitions. The complex concurrent computations of the system so axiomatized then exactly correspond to proofs in the logic. Maude specifications can be efficiently executed using the Maude rewrite engine [CELM96], thus allowing their use for system prototyping and the debugging of specifications.

Rewriting logic has proved to be a general and flexible semantic framework for specifying a wide variety of models of concurrency [Mes92, Mes96b], and in particular for specifying concurrent object systems [Mes93]. In this paper, given that a concurrent object-oriented style fits communication protocols very well, we will focus on the specification of concurrent object-oriented systems.

The main object-oriented features of Maude are class definitions, subclass declarations, message declarations, and rewrite rules axiomatizing the system's semantics. A class definition `class C | att1: s1 ...attn: sn` defines a class of name `C` with attribute names `att1, ..., attn` and their corresponding sorts. Objects of such a class are then record-like structures of the form `< O; C | att1: v1 ... attn: vn >` where `O` is the name of the object, and `v1 ...vn` the current values of its attributes. Objects can interact with each other in a variety of ways, including the sending of messages. A message declaration `msg m : p1 ... pn -> Message` defines the name of the message and the sorts of the set of parameters. Often, the first parameter refers to the object the message is sent to. The state of a concurrent object system is called a *configuration*. Typically, a configuration is a multiset of objects and messages. In Maude, the multiset union operator for configurations is associative and commutative—so that order and parentheses do not matter, and so that rewriting is multiset rewriting—and is denoted with empty syntax (juxtaposition). The dynamic behavior of a concurrent object system is then axiomatized by specifying each of its basic concurrent transition patterns by a corresponding labeled rewrite rule. Each such rewrite rule rewrites a given configuration `C` to another configuration `C'`, perhaps subject to a given condition `cond` being satisfied. This is expressed with the syntax `crl [label] : C => C' if cond`. An unconditional rule is expressed with the syntax `rl [label] : C => C'`. For example, the following unconditional rewrite rule

```
rl [rule1] : m1(O1,O3) < O1; C1 | a1: v1, a2: n >
```

```
=> < O1; C1 | a1: f(v1), a2: n+1 >
   < O1.n; C2 | b: g(v1) > m2(O3)
```

expresses the result of an asynchronous message reception of type `m1` by an object of class `C1`, namely, the message is consumed, the object changes its `a1` value to `f(v1)` and has its `a2` value increased by one, a new object of class `C2` with identity `O1.n` is created, and a message of type `m2` is sent to `O3`. In general, a rewrite rule may have several objects in its lefthand side, describing a *synchronization* or *multi-party interaction* between several objects. However, in most of the rules in the communication protocol examples that we shall discuss the communication will be asynchronous and the lefthand sides will consist of either an object and a message, or just an object. By convention, attributes whose value does not change and does not affect the next state of other attributes need not be mentioned in a rule.

3 Specifying and Executing the NSPK Protocol

The NSPK Protocol [NS78] is a cryptographic protocol for the authentication of pairs of agents in a distributed computer system. For instance, agents want to be assured of each other's identity before they exchange security-critical data. Thus, an intruder should not be allowed to impersonate another agent. For this purpose, initiator and responder of a communication mutually authenticate each other. NSPK uses public key cryptography, i.e., each agent possesses a public key which can be accessed by all agents and a secret key, which is the inverse of the public key. Moreover, nonces are used in the protocol. Nonces are freshly generated, random numbers to be used in a single run of the protocol. It is assumed that these numbers are generated in such a way that they cannot be guessed by other agents. A textbook-style simplified description of NSPK as for instance given in [Low96] is:

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

In this protocol A tries to establish a communication with B. Thus, A plays the role of the initiator and B reacts in the responder role. A encrypts his nonce N_a with the public key of B and sends it along with his identity to B. Now A is in a waiting status. B decrypts the message, creates a new nonce and sends back to A the concatenation of both nonces encrypted with the public key of A. After this, B is waiting for the answer from A. A can decrypt the message sent by B. In case the contents of the message is the concatenation of his own nonce N_a plus another nonce, A can be assured that the message was sent by B, because B is the only agent who can decrypt the message sent by A containing the nonce N_a. Thus, A establishes a communication with B. Moreover, to acknowledge to B the receipt of B's nonce, A separates it from the message and sends it back. If B receives this message from A, he can be assured of being talking to A and, therefore, of having established a communication with A.

As one can see, there is a lot of implicit knowledge in the textbook-style specification of the NSPK Protocol. Moreover, there are open questions concerning how agents behave if they receive a message which does not have the expected contents, e.g., if A receives a message from B which does not contain his nonce.

Our intention is to make this implicit knowledge explicit and to clarify fuzzy parts of the description by formalizing the protocol in rewriting logic. In addition to the protocol requirements given above, we will provide a Maude specification which allows multiple sessions for an agent. For instance, an agent may simultaneously participate in several runs of the protocol. In this way, an agent may simultaneously play different roles (e.g., initiator or responder) in different runs.

We need to provide some application-specific data types such as fields, nonces, keys, etc. These types are equationally specifiable with the help of Maude's functional modules.

```
fmod DATATYPES is
  sorts Key Field FieldSet Nonce Principal Run Role EstabComm .
  subsort Nonce Principal Key < Field .
  subsort Field < FieldSet .
  op keypair : Key Key -> Bool [comm] .
  op mtfield : -> Field .
  op ped : Key Field -> Field .
  op n : Principal Nat -> Nonce .
  op cat : Nonce Nonce -> Field .
  op (_,_,_) : Nonce Principal Field  -> Run .
  op (_,_,_,_) : Role Nonce Principal Nonce -> EstabComm .
  ops i r : -> Role .
  vars sk pk : Key . var f : Field .
  ceq ped(sk,ped(pk,f)) = f  if keypair(sk,pk) .
  ...
```

In the sort declaration, application-specific data types such as keys, fields, nonces, principals (i.e., identities of agents), etc., are defined. Sorts may be ordered hierarchically by defining subsort relationsships. For instance, nonces, principals, and keys are understood as specialized fields. The operation `keypair` is true for pairs of secret and public keys of one agent. `ped`[1] is the operation to encrypt and decrypt fields. Agents send data of type `Field` in a message. `mtfield` is a generator function for the sort `field`. `n` is a function to generate fresh, cryptographic nonces. To make theses nonces unique, we implemented this function by using the address of an agent and a number. Concatenation of two nonces is denoted with `cat`. The conditional equation defines the behavior of `ped`. In the complete specification appropriate equations for all operations are provided. We intend to use the sort `Run` to store information about different sessions of an agent. An agent may concurrently communicate with several other agents by establishing several runs. The first parameter of the run-3-tuple is

[1] "public key encryption and decryption" (borrowed from CAPSL). This operator behaves like RSA encryption.

the nonce which is newly created by the agent for the session. The second and third arguments are the identity and nonce of the communication partner. We assume that nonces are always freshly generated and, therefore, can be used to uniquely identify a run. EstabComm will be used to store information about successfully established communications, in either of the two roles, initiator or responder. For each established communication an agent stores his own nonce, and for the communication partner it stores the identity and the nonce.

We define an object class Agent in Maude. Agents will model the legal participants of a protocol. Later on we will also define a class Intruder. Agents have a secret key, two attributes storing information about possible runs, one for each possible role, an attribute to store information about established communications, an attribute for desired communications, i.e., a set of object identifiers to identify the agents they want to communicate with, and a counter which will be used to generate nonces.

```
class Agent | e_com: EstabCom sec_key: Key role_i: Run role_r: Run
              d_com: FieldSet cnt: Nat .
```

Moreover, we need one message to send data between agents. The first two parameters of this message contain the identities of the sender and receiver agents. The third argument is the content to be sent, i.e., an encrypted field. Therefore, in contrast to the specification in [Low96], it is not necessary to include sender and receiver identities in the field.

```
msg from_to_send_ : Principal Principal Field  -> Message .
```

The Maude specification of NSPK consists of six rules. The first rule BeginRun defines the start of a protocol run. The second rule Message1Rec defines the behavior of a responder agent to a request of starting a protocol session.

```
vars A B P : Principal . vars RI RR : Run . vars NI : Nonce .
var F : Field . var S : FieldSet . vars SKB PKB SKA PKA : Key .
var C : EstabCom . var J : Nat ....

rl [BeginRun] :
  < A; Agent | role_i: RI, d_com: B U S, cnt: J >
  => < A; Agent | role_i: RI U (n(A,J),B,mtfield), d_com: S, cnt: J + 1 >
     from(A)to(B)send(ped(pk(B),n(A,J))) .

crl [Message1Rec] :
  < B; Agent | sec_key: SKB, role_i: RI, role_r: RR, cnt: J >
  from(A)to(B)send(ped(PKB,F))
  => < B; Agent | role_r: RR U (n(B,J),A,F), cnt: J + 1 >
     from(B)to(A)send(ped(pk(A),cat(F,n(B,J))))
  if keypair(SKB,PKB) and not(F in RR) .
```

If an agent A wants to talk to at least one other agent, say B (U is the union operator for field sets), then he starts a protocol run by sending to this agent his address A and a newly created nonce n(A,J). He deletes the agent B from his set of desired communication partners and keeps track of the information of the protocol run in attribute `role_i`. Additionally, the agent encrypts his nonce `n(A,J)` with the public key of his communication partner[2].

If an agent gets a request for a protocol session, he stores the information in attribute `role_r` and sends back the encrypted concatenation of nonces, provided he could decrypt the message. The second part of the condition `not(F in RR)` assures that agent `B` is not already communicating with `A` in a responder role. As already mentioned, by default convention one only needs to mention in the righthand side of the rule those attributes which change their value.

The next two rules deal with the receipt of Message 2. In case the initiator gets back his own nonce plus the new nonce of the other agent, he establishes the communication and triggers the last message, i.e., sending Message 3. This behavior is specified in rule `Message2RecCorrect`. Otherwise, the initiator refuses to establish the communication, does not send back any message and just stops the current protocol run (`Message2RecIncorrect`).

```
crl [Message2RecCorrect] :
  < A; Agent | sec_key: SKA, role_i: RI U (NI,P,mtfield), e_com: C >
  from(B)to(A)send(ped(PKA,F))
  => < A; Agent | role_i: RI, e_com: C U (i,NI,B,rest(F)) >
     from(A)to(B)send(ped(pk(B),rest(F)))
  if keypair(SKA,PKA) and (B == P) and NI == first(F)  .

crl [Message2RecIncorrect] :
  < A; Agent | sec_key: SKA, role_i: RI U (NI,P,mtfield) >
  from(B)to(A)send(ped(PKA,F))
  => < A; Agent | role_i: RI >
  if keypair(SKA, PKA)  and NI == first(F) and B =/= P .
```

`first` and `rest` are operations which are defined on fields. Applied to a concatenation of nonces they deliver the first or last one, respectively. Analogously, there are two rules for the responder covering the cases for the correct and the incorrect receipt of Message 3 of the NSPK Protocol. Because of space limitations we refrain from presenting these rules. They are very similar to the last two rules above.

A slightly extended version of the specification presented above has been executed on the Maude rewrite engine [CELM96]. The given initial configuration involved two agents `Alice` and `Bob`, where `Alice` wants to talk to `Bob`. Using the default rewriting strategy we already found an unexpected behavior. Analyzing the rewrite path showed that the problem was connected with the ability of having several runs per agent. The condition of the rule `Message1Rec` was not sufficient. If the responder sends Message 2 of the NSPK Protocol, i.e., `from(Bob)to(Alice)send(ped(pk(Alice),cat(NI,n(Bob,J))))`, the rule `Message2RecCorrect` should be applied to indicate the situation where `Alice` already started a run with `Bob` and she waits for his answer. But there is also a matching for another rule, namely `Message1Rec`. Thus, `Alice` sets up a new

[2] `pk(B)` is understood as a function which delivers the public key of agent `B`. In our implementation we used a table of public keys which is accessible by all agents.

responder session with `Bob` as initiator. The correct condition for the rule with label `Message1rec` is `if keypair(SKB,PKB) and not(F in RR) and not(first(F) in RI)`. This example illustrates the usefulness of executing specifications in the Maude engine for validation purposes.

4 The Intruder and an Attack

The intruder is part of the system and, therefore, may participate in normal protocol runs. Moreover, an intruder may observe, intercept or fake messages. Introducing a fake message into the system either means replaying a previously observed or intercepted message, or creating a new message using the nonces known by the intruder. In Maude we can give a comprehensive specification of a general intruder who is able to perform all the mentioned actions.

An intruder has usually all attributes an agent has plus additional ones to store the set of nonces he knows and all the messages he observed or intercepted.

```
class Intruder | e_com: EstabCom sec_key: Key ncs: FieldSet,
                 msgs: FieldSet agents: FieldSet role_i: Run,
                 role_r: Run d_com: Field cnt: Nat.
```

Moreover, since the intruder may behave as a normal agent, all agent rules are redefined in such a way, that whenever an intruder creates or receives a nonce he stores it in corresponding attribute `ncs`, similarly, remembers all seen agent identities in `agents`. We do not give all these redefined rules. They are very similar to the normal agent rules, except for being extended so that the intruder can remember nonces and agent identities. Instead we describe the five rules which cover the intrusion behavior.

```
crl [IntruderInterceptMessage] :
  < I; Intruder | sec_key: SKI, ncs: N, msgs: M, agents: S >
  from(B)to(A)send(ped(KEY,F))
  => if keypair(KEY,SKI)
     then < I; Intruder | ncs: N U F, agents: S U A U B >
     else < I; Intruder | msgs: M U ped(KEY,F), agents: S U A U B > fi
  if A =/= I and B =/= I .
```

If the intruder intercepts a message that is encoded with his key, he can decrypt it and store the nonce in his list of nonces. A similar rule for overhearing messages only differs from the interception rule in the way the messages remain in the configuration. The following two rules express the intruder's ability to fake messages.

```
crl [IntruderReplayMessage] :
  < I; Intruder | msgs: M U MES, agents: S U A U B >
  => < I; Intruder | msgs: M U MES > from(B)to(A)send(m)
  if MES =/= mtfield and A =/= I .

crl [IntruderFakeMessage] :
  < I; Intruder | ncs: N U F, agents: S U A U B >
  => < I; Intruder | ncs: N U F > from(A)to(B)send(ped(pk(B),F))
  if B =/= I .
```

To prove the correctness of the protocol one has to prove: (1) the authentication of the responder, i.e., that an initiator only establishes a communication with another agent if the agent took part in the protocol run, and (2) the authentication of the initiator. Concerning item (2) there is a well-known attack to the NSPK Protocol proposed by Gavin Lowe [Low96]. This attack is newer and more subtle than the one proposed by Denninger and Sacco in [DS81] which allows an intruder to replay old, compromised public keys. Again, we can take advantage of the Maude engine to analyze the enriched specification "NSPK+Intruder". Running this specification on the Maude engine using a breadth-first-search strategy delivers the attack given in [Low96].

5 A Breadth-First-Search Strategy

We have used the Maude rewrite engine [CELM96] to validate the specification "NSPK+Intruder". Thanks to the reflective nature of rewriting logic [CM96], it is possible to express strategies for executing Maude specifications in Maude itself. This is particularly important because Maude specifications formalize concurrent systems that can be highly concurrent and nondeterministic, and that may not terminate. Therefore, in addition to a default execution strategy that ensures a fair application of the rewrite rules, it is possible to define a very wide variety of rewriting strategies using rewrite rules at the metalevel, and to then execute a specification with a given strategy in Maude. For example, one can program a breadth-first-search strategy to exhaustively determine all possible behaviors from a given initial state using the rewrite rules of a Maude specification. In this way we can either validate the specification and detect errors, or do formal reasoning about the specification. To analyze behaviors starting from a symbolic description of a set of states, the technique of narrowing can instead be used (see Section 6).

For the "NSPK+Intruder" specification we implemented a breadth-first-search strategy which searches for possible attacks. To efficiently implement this strategy, we have chosen to use a bounded depth-first-search strategy which is recursively called with increasing bounds as long as no attack is discovered. A breadth-first-search strategy which starts at a given depth and iteratively calls a depth-first search with increased maximum depth is given as follows:

```
var StartDepth : MachineInt .
op breadthFirstSearchUntilStopRules : Module Term QidList QidSet
                                      MachineInt -> Strategy .
op boundedDepthFirstSearchUntilStopRules : Module Term QidList QidSet
                                           MachineInt -> Strategy .
eq breadthFirstSearchUntilStopRules(M, T, Labels, StopLabels, StartDepth)
 = if boundedDepthFirstSearchUntilStopRules(
            M, T, Labels, StopLabels, StartDepth) == stop(0, emptyPath)
   then boundedDepthFirstSearchUntilStopRules(
            M, T, Labels, StopLabels, StartDepth + 1)
   else boundedDepthFirstSearchUntilStopRules(
            M, T, Labels, StopLabels, StartDepth)    fi .
```

Both strategies have five parameters: a module with the specification, a term which is the initial configuration, a list of labels which should be applied during the computation (i.e., all labels in the specification), a set of labels for which the search stops, and a depth, which in the case of the breadth-first-search strategy is the depth to start with, and in the case of the bounded depth-first-search strategy is the maximum depth.

We added a rule to the specification which can only be applied if an attack occurred. One possible definition of a successful attack is the following: An successful attack is determined by the fact that nonces have been compromised, i.e., an intruder possesses two nonces with which another agent established a communication. The conditions of the following rule assure that it can only be applied in case an attack happened. The intruder remembers the successful attack by storing the information in the set of established communications.

```
crl [IntruderRecognizeAttack] :
  < I; Intruder | e_com: EC, role_i: RI, role_r: RR, ncs: N U N1 U N2 >
  < A; Agent | e_com: EC' U (ROLE,N1,B,N2) >
  => if ROLE == i
     then < I; Intruder | e_com: EC U (r,N2,A,N1) >
          < A; Agent | e_com: EC' U (ROLE,N1,B,N2) >
     else < I; Intruder | e_com: EC U (i,N2,A,N1) >
          < A; Agent | e_com: EC' U (ROLE,N1,B,N2) > fi
  if not((r,N2,A,N1) in EC) and not((i,N2,A,N1) in EC) and not(N1 in RR)
     and not(N1 in RI) and not(N2 in RR) and not(N2 in RI) and B =/= I .
```

We implemented a bounded depth-first-search strategy which, given a maximum depth and a set of rule labels, searches for a computation path (less or equal in length to the given maximum depth) in which one of the rules whose label is in the given stop set of labels is applied. The semantics of this strategy is given by a set of equations using some auxiliary functions. The full algorithm is given in the Appendix A. The intuitive idea is that, starting in the initial configuration, for a rewrite step all rules in the given list of rules are tried until a rewrite is found. In case there is no path containing the label in the given set, the empty path is returned; otherwise, the first such path found in the search is returned. A rewrite step (`step`) consists of a label, a number indicating which substitution for the rule with that label was used, and a term that is precisely the righthand side replacement for the given label, rule, and substitution. Thus, a path of the form `path(T1, step(L1,N1,T2), step(L2,N2,T3))` represents a computation sequence which starts in configuration `T1` and in this configuration the rule with label `L1` and matching substitution number `N1` has been applied and resulted in configuration `T2`, from which rule `L2` could be applied with the `N2`th matching substitution resulting in configuration `T3`.

The bounded depth-search-first algorithm continues building the path, provided that none of the rules in the chosen stop set are applied, nor the maximum depth is reached. For each new step in the path all rewrite rules are tried. Whenever a path is closed, i.e., whenever no other rewrite rules are applicable, or the maximum depth is reached, the algorithm backtracks. Backtracking means that first the same rule as in the previous step is tested with the next matching substitution number. If this fails, then the next rule in the list of applicable rules is tried. If this also fails, another backtracking step is taken until no more rules can be applied from the initial configuration. If any of the previous two attempts is successful, the bounded depth-first-search algorithm for the new, extended path is called.

Applying the depth-first-search strategy to the implemented specification and depth 11 results in the attack which has been described in [Low96]. This result is achieved on a 300Mhz Linux 4.2 PC in less than 96 minutes involving more than 28 million rewrites. For details see Appendix A.

6 Future Developments

Our future research plans will be along the following three main lines: (1) narrowing using symbolic execution techniques, (2) modularity and compositionality issues, (3) combination of rewriting logic and temporal logic.

6.1 Narrowing

Our systematic search for an attack using a breadth-first strategy has illustrated a form of model checking in which we explore all the possible behaviors beginning with a given state using the reflective capabilities of Maude. The fact that state-building operators—such as the multiset union of configurations—obey structural axioms like associativity and commutativity allows the rewriting to take place *modulo* such axioms. This means that we have a *single* representation for the many different expressions describing such a state, and therefore a much smaller search space than if rewriting modulo axioms were not supported.

But what about exploring behaviors starting not just from one state, but from a, possibly infinite, *set* of states? For object-oriented protocol specifications of the general style illustrated by the NSPK example, this can be accomplished by performing *narrowing* [Fay79] modulo structural axioms such as the associativity and commutativity of the multiset union of configurations. The idea is to describe a set of initial states by a symbolic expression with variables that stands for the set of all its ground substitutions. Then, the exploration of behaviors is entirely similar to the rewriting exploration, except that at each step, instead of *matching* the lefthand side of a rule to a ground term representing a single state, we *unify* (modulo structural axioms such as associativity and commutativity) such a lefthand side with a nonvariable subterm of the symbolic expression representing the current set of states. In principle, such narrowing and unification algorithms can be specified by means of rewrite rules in Maude, but for efficiency reasons it is better to build in the unification algorithm.

We plan to add to Maude built-in unification modulo several equational theories and their combinations, and to exploit such a facility to explore behaviors for sets of states by narrowing. There are some similarities between our proposed use of narrowing and the narrowing analysis performed in the NRL Protocol Analyzer [Mea95, Mea96a, Mea96b]. But in the NRL tool narrowing (with standard unification) is performed only by means of the Church-Rosser *equations* that axiomatize the basic algebraic properties of the underlying cryptography, whereas our proposed use is different, namely, to perform narrowing (modulo structural axioms) using both the (Church-Rosser) equations and the (in general non-Church-Rosser) transition rules specifying the system as a rewrite theory.

6.2 Towards Modular and Dynamic Protocols

In practice NSPK and other protocols are services designed to ensure desired communication semantics or other properties of the runtime infrastructure. They are not—as it might appear from the usual discussions—the main objective of the agents involved. Typically, such protocols are hardwired into application code, making the code both complex and brittle. It is important to be able to specify and implement functionality and protocols in a *modular* way, so that protocols can be installed dynamically and can be composed in many ways to meet complex and changing requirements. Composition of system properties is in general a nontrivial matter (c.f. [Pel94, Rus94, Rus95]) and a modular approach seems particularly promising in this regard. Subclassing is one way to obtain some degree of modularity. An alternative is to use reflection to achieve a highly expressive, modular, dynamic formalism. We sketch very briefly the main ideas. We want operations that compose two or more protocols to form a new protocol, and than install a protocol on a system to obtain a new system. These operations should work on specifications, code, or running systems. Let $\mathcal{S}[\mathcal{C}]$ denote the installation of protocol $\mathcal{S}$ on system $\mathcal{C}$. Using the initial semantics of Maude it is easy to formalize notions such as $\mathcal{S}$ ensures property P for $\mathcal{C}$ (written $\mathcal{S}[\mathcal{C}] \models_{\mathrm{ind}} P$). Two key questions are: if $\mathcal{S}$ ensures property P for $\mathcal{C}$ under what conditions does $\mathcal{S}$ ensure property P for $\mathcal{S}'[\mathcal{C}]$; if $\mathcal{S}$ ensures property P for $\mathcal{C}$ and $\mathcal{S}'$ ensures property P' for $\mathcal{C}$ under what conditions does $\mathcal{S}$ ensure P' for $\mathcal{S}'[\mathcal{C}]$. Reflection can be used as a mechanism for installing protocols by composing a protocol specified as a meta-level activity and a system specified at the base level. Two reflective architectures have been developed for the actor model and used to provide a basis for defining and reasoning about dynamic adaptable distributed systems: the *onion skin* model [Agh97] aims at layered specification of interaction policies for components of distributed systems and allows different protocols for security, reliability, quality of service and other policies to be composed and modified dynamically as the system runs and new needs arise by adding new meta-levels; the *two level actor model* [Ven95] focuses on horizontal composition of system-wide services provided by meta-actors that manage resources and control runtime properties of base level application components in an open-distributed system. These reflective models can be naturally represented in Maude and used to attain a high degree of modularity and dynamicity.

6.3 Combining Rewriting and Temporal Logic

Rewriting logic is a logic that we could call *internal* to the concurrent computation of a system that it axiomatizes. Deduction exactly corresponds to computation, and in fact can be used to implement the system axiomatized by the rules. By contrast, temporal logic can be regarded as taking an *external* view of the system. It regards it as a mathematical model—typically some kind of Kripke structure—about which it then makes assertions about its global properties such as safety or liveness. Both levels of description and analysis are useful in their own right; in fact, they complement each other. We therefore plan to use both logics in combination to prove properties about communication protocols and other concurrent systems.

The first issue that must be addressed is the semantic integration between the two logics. Essentially, such an integration is straightforward, because both logics are talking about the same mathematical model. Theories in rewriting logic have *initial models*. The initial model $\mathcal{T}_{\mathcal{R}}$ of a rewrite theory $\mathcal{R}$ is a mathematical model of the concurrent system axiomatized by $\mathcal{R}$. Specifically, it is a category with algebraic structure [Mes92], where the objects correspond to system states, and the arrows correspond to concurrent system transitions. Therefore, $\mathcal{T}_{\mathcal{R}}$ can be regarded as a Kripke structure whose transitions are labeled by the arrows of the category. A variety of different modal or temporal logics can then be chosen to make assertions about such a Kripke structure. We are particularly interested in temporal logics of this kind that provide explicit support for objects. Two candidates that we are considering are the version of the μ-calculus proposed by Ulrike Lechner [LL96] for reasoning about object-oriented Maude specifications, and the object-oriented Distributed Temporal Logic (DTL) of Ehrich et al. [ECSD98, DE97]. For the second alternative a temporal logic extension of rewriting logic and a technique to transform a rewrite theory into a temporal logic theory have been proposed in [Den98]. In future work we will investigate how one can reason in the combined calculus and how such reasoning can be supported by tools.
Other approaches to reason about security protocols include Gray and McLean's [GM95], who use Lamport's TLA to specify cryptographic protocols and correctness requirements, and Syverson and Meadows' [Syv93], who propose a logical language for specifying protocol requirements and translate them by hand into the Prolog language of the NRL Analyzer [Mea95, Mea96a, Mea96b]. There is also a large body of work by various authors using *abstraction* techniques to reduce communication protocols—that in principle have an infinite number of states—to finite-state versions for which temporal logic formulas can be decided by model checking. We conjecture that abstraction maps will have a natural high-level specification in rewriting logic as adequate maps between theories.

Our medium-term goal is to reach high assurance for secure communication protocols by combining the advantages of Maude executable specification, formal analysis by rewriting-based model checking and narrowing, and temporal logic reasoning by means of both theorem proving and finite-state model checking using abstractions. We expect that rewriting logic and Maude will provide a good logical framework for the integration and implementation of these techniques.

Acknowledgments

We thank Jonathan Millen for very helpful discussions on secure protocols and NSPK. We appreciated very much his valuable comments on an earlier version of this paper. We thank Francisco Durán for his kind help at different stages of this research, particularly on Maude matters, and Ulrike Lechner and Narciso Martí-Oliet for many discussions on combining temporal and rewriting logic. We are grateful to Hans-Dieter Ehrich for discussion on the temporal logics for asynchronous communication.

References

[Agh97] G. Agha. Abstracting interaction patterns: A programming paradigm for open distribute systems. In H. Bowman and J. Derrick, editors, *Formal Methods for Open Object-Based Distributed Systems (FMOODS'97), Volume 2, IFIP TC6 WG6.1 Intern. Workshop, 21-23 July, Canterbury, Kent, UK*, pages 135–153. Chapman & Hall, 1997.

[CELM96] M. Clavel, S. Eker, P. Lincoln, and J. Meseguer. Principles of Maude. In Meseguer [Mes96a], pages 65–89.

[Cla98] Clavel, M. and Durán, F. and Eker, S. and Meseguer, J. Building Equational Proving Tools by Reflection in Rewriging Logic, 1998.

[CM96] M. Clavel and J. Meseguer. Reflection and Strategies in Rewriting Logic. In Meseguer [Mes96a], pages 125–147.

[DE97] G. Denker and H.-D. Ehrich. Specifying Distributed Information Systems: Fundamentals of an Object-Oriented Approach Using Distributed Temporal Logic. In H. Bowman and J. Derrick, editors, *Formal Methods for Open Object-Based Distributed Systems (FMOODS'97), Volume 2, IFIP TC6 WG6.1 Intern. Workshop, 21-23 July, Canterbury, Kent, UK*, pages 89–104. Chapman & Hall, 1997.

[Den98] G. Denker. From Rewrite Theories to Temporal Logic Theories: A Distributed Temporal Logic Extension of Rewriting Logic. Manuscript; SRI International, Menlo Park, CA, 1998.

[DM98] F. Durán and J. Meseguer. An Extensible Module Algebra For Maude. Manuscript, SRI International, May, 1998.

[DS81] D. E. Denning and G. M. Sacco. Timestamps in Key Distribution Protocols. *Communications of the ACM*, 24(8):533–536, 1981.

[ECSD98] H.-D. Ehrich, C. Caleiro, A. Sernadas, and G. Denker. Logics for Specifying Concurrent Information Systems. In J. Chomicki and G. Saake, editors, *Logics for Databases and Information Systems*, pages 167–198. Kluwer Academic Publishers, 1998.

[Fay79] M. Fay. First-order unification in an equational theory. In *Proceedings of the 4th Workshop on Automated Deduction*, pages 161–167, 1979.

[GM95] J. W. Gray and J. D. McLean. Using Temporal Logic to Specify and Verify Cryptographic Protocols (Progress Report). In *Proc. of the 8th IEEE Computer Security Foundations Workshop*. IEEE Computer Society Press, 1995.

[LL96] U. Lechner and C. Lengauer. Modal–μ–Maude — specification and properties of concurrent objects. In B. Freitag, C.B. Jones, C. Lengauer, and H.-J. Schek, editors, *Object Orientation with Parallelism and Persistence*, pages 23–46. Kluwer, 1996.

[Low96] G. Lowe. Breaking and Fixing the Needham-Schroeder Public-Key Protocol Using FDR. In *Second Intern. Workshop TACAS'96, Passau, Germany, March 1996*, pages 147–166. Springer, 1996. LNCS 1055.

[Mea95] C. A. Meadows. Formal Verification of Cryptographic Protocols: A Survey. In *Advances in Cryptology - Asiacrypt '94*, pages 133–150. Springer, 1995. LNCS 917.

[Mea96a] C. A. Meadows. Analyzing the Needham-Schroeder Public Key Protocol: A Comparison of Two Approaches. In *Proc. of ESORICS'96*. Springer, 1996. LNCS.

[Mea96b] C. A. Meadows. Language Generation and Verification in the NRL Protocol Analyzer. In *Proc. 9th Computer Security Foundations Workshop*. IEEE Computer Society Press, June 1996.

[Mes92] J. Meseguer. Conditional Rewriting Logic as a Unified Model of Concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.

[Mes93] J. Meseguer. A Logical Theory of Concurrent Objects and its Realization in the Maude Language. In G. Agha, P. Wegner, and A. Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*, pages 314–390. The MIT Press, 1993.

[Mes96a] J. Meseguer, editor. *Rewriting Logic and its Applications, First International Workshop, Asilomar Conference Center, Pacific Grove, CA, September 3-6, 1996*. Elsevier Science B.V., Electronic Notes in Theoretical Computer Science, Volume 4, `http://www1.elsevier.nl/mcs/tcs/pc/volume4.htm`, 1996.

[Mes96b] J. Meseguer. Rewriting Logic as a Semantic Framework for Concurrency: A Progress Report. In U. Montanari and V. Sassone, editors, *Proc. 7th Intern. Conf. on Concurrency Theory: CONCUR'96, Pisa, August 1996*, pages 331–372, 1996. LNCS 1119.

[Mes98] J. Meseguer. Research Directions in Rewriting Logic. In U. Berger and H. Schwichtenberg, editor, *Computational Logic, NATO Advanced Study Institute, Marktoberdorf, Germany, July 29 – August 6, 1997*. Springer Verlag, 1998. *To appear.*

[Mil97] J. Millen. CAPSL: Common Authentication Protocol Specification Language. Technical Report MP 97B48, The MITRE Corporation, 1997.

[NS78] R. Needham and M. Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21(12):993–999, 1978.

[Pel94] J. Peleska. On a unified formal approach for the development of fault-tolerant and secure systems. In H. Rischel, editor, *Nordic Seminar on Dependable Computing Systems, Lyngby, Denmark, August 1994. Technical University of Denmark*, pages 69–80, 1994.

[Rus94] John Rushby. Critical system properties: Survey and taxonomy. *Reliability Engineering and System Safety*, 43(2):189–219, 1994.

[Rus95] John Rushby. Combining system properties: A cautionary example and formal examination. Technical report, Computer Science Laboratory, SRI International, Menlo Park, CA, June 1995. Unpublished project report; available at `http://www.csl.sri.com/~rushby/combined.html`.

[Syv93] Syverson, P. and Meadows, C. A. A Logical Language for Specifying Cryptographic Protocol Requirements. In *Proc. IEEE Computer Society Symp. on Research in Security and Privacy, Oakland, CA*, pages 165–177, May 1993.

[Ven95] Venkatasubramanian, N and Talcott, C. L. Reasoning about Meta Level Activities in Open Distributed Systems. In *Principles of Distributed Computation*. ACM, 1995.

A Search Strategy

The module `META-LEVEL` provides several sorts such as list of identities (`QidList`), integer (`MachineInt`), terms (`Term`), modules (`Module`), etc. as well as several operations on these sorts, e.g., `op nilQidList : -> QidList`, and `op qidList: QidList QidList -> QidList [assoc id: nilQidList]`. For details about Maude's meta-level the reader is referred to the Maude manual [Cla98].

The Bounded Depth-first-search Strategy

```
fmod MOD is
protecting META-LEVEL .

sorts Path Step Strategy .
subsort Term < Path .

var Labels : QidList .
var StopLabels : QidSet .
var APath : Path .
var QI QI' L : Qid .
vars QIList QIList' : QidList .
var N : MachineInt .
var T : Term .
var M : Module .

op step : Qid MachineInt Term ->  Step .
op noStep : -> Step .
op path : Path Step -> Path .
op emptyPath : -> Path .
op lastTerm : Path -> Term .
op rest : Qid QidList -> QidList .
op memberQidQidSet : Qid QidSet -> Bool .

eq rest(QI, qidList(qidList(qidList(QIList, QI), QI'), QIList'))
   = qidList(QI', QIList') .
eq rest(QI, qidList(QIList, QI)) = nilQidList .
```

```
eq lastTerm(path(APath, step(L, N, T))) = T .
eq lastTerm(T) = T .
eq memberQidQidSet(QI, qidSet(QI,SS)) = true .

op stop : MachineInt Path -> Strategy .
op depthFirstSearchUntilStopRules : Module Term QidList QidSet
                                    MachineInt -> Strategy .
op depthFirstSearchUntilStopRulesAux : Module MachineInt Path QidList
                                       QidSet MachineInt -> Strategy .
op backtrack : Module MachineInt Path QidList QidSet MachineInt -> Strategy .
op nextRewrite : Module Term QidList -> Step .

eq depthFirstSearchUntilStopRules(M, T, Labels, StopLabels, MaxDepth)
 = depthFirstSearchUntilStopRulesAux(M, 0, T, Labels, StopLabels, MaxDepth) .

eq depthFirstSearchUntilStopRulesAux(M, 0, T, Labels, StopLabels, MaxDepth)
  = if nextRewrite(M, T, Labels) == noStep
    then stop(0, emptyPath)
    else depthFirstSearchUntilStopRulesAux(
            M, 1, path(T, nextRewrite(M, T, Labels)),
            Labels, StopLabels, MaxDepth)
    fi .

eq depthFirstSearchUntilStopRulesAux(
      M, Depth, path(APath, step(L, N, T)),
      Labels, StopLabels, MaxDepth)
  = if memberQidQidSet(L, StopLabels) == true
    then stop(Depth, path(APath, step(L, N, T)))
    else if Depth < MaxDepth
         then if nextRewrite(M, T, Labels) == noStep
              then backtrack(M, Depth,  path(APath, step(L, N, T)),
                             Labels, StopLabels, MaxDepth)
              else depthFirstSearchUntilStopRulesAux(
                     M, Depth + 1,
                     path(path(APath, step(L, N, T)), nextRewrite(M, T, Labels)),
                     Labels, StopLabels, MaxDepth)
              fi
         else backtrack(M, Depth, path(APath, step(L, N, T)),
                        Labels, StopLabels, MaxDepth)
         fi
    fi .

eq backtrack(
         M, Depth, path(APath, step(L, N, T)),
         Labels, StopLabels, MaxDepth)
  = if meta-apply(M, lastTerm(APath), L, emptySubstitution, N + 1)
         == error*, emptySubstitution
    then if nextRewrite(M, lastTerm(APath), rest(L, Labels)) == noStep
         then backtrack(M, Depth - 1, APath,
                        Labels, StopLabels, MaxDepth)
```

```
        else depthFirstSearchUntilStopRulesAux(
               M, Depth,
               path(APath,
                    nextRewrite(M, lastTerm(APath), rest(L, Labels))),
               Labels, StopLabels, MaxDepth)
        fi
    else depthFirstSearchUntilStopRulesAux(
          M, Depth,
          path(APath, step(L, N + 1,
                           extTerm(meta-apply(M, lastTerm(APath), L,
                                              emptySubstitution, N + 1)))),
          Labels, StopLabels, MaxDepth)
    fi .

eq backtrack(M, 0, T, Labels, StopLabels, MaxDepth)
  = stop(0, emptyPath) .

eq nextRewrite(M, T, nilQidList)
  = noStep .

eq nextRewrite(M, T, qidList(L, Labels))
  = if meta-apply(M, T, L, emptySubstitution, 0)
        == error*, emptySubstitution
    then nextRewrite(M, T, Labels)
    else step(L, 0, extTerm(meta-apply(M, T, L, emptySubstitution, 0)))
    fi .

op NSPK : -> Module .
eq NSPK = NSPK .
endfm
```

$\overline{\texttt{NSPK}}$ is the meta-level representation of the NSPK protocol using an object oriented extension of `META-LEVEL` [DM98].

Result of the Breadth-first Search

We reduce the term

```
red breadthFirstSearchUntilStopRules(NSPK,'cf2Agents1Intruder,
    qidList('IntruderRecognizeAttack, 'BeginRun, 'IntruderBeginRun,
            'Message1Rec, 'Message2RecCorrect, 'Message3RecCorrect,
            'IntruderAcceptEveryMessage, 'IntruderInterceptMessage,
            'IntruderFakeMessage, 'IntruderReplayMessage,
            'IntruderMessage1Rec, 'IntruderMessage2RecCorrect,
            'IntruderMessage3RecCorrect, 'Message2RecIncorrect,
            'Message3RecIncorrect, 'IntruderMessage2RecIncorrect,
            'IntruderMessage3RecIncorrect, 'IntruderOverhearMessage),
    'IntruderRecognizeAttack, 11) .
```

for the given initial configuration

```
eq conf2Agents1Intruder =
   < alice; Agent | e_com: mtecom, sec_key: skalice, role_i: mtrun,
                    role_r: mtrun, d_com: mrx, cnt: 0 >
   < bob; Agent | e_com: mtecom, sec_key: skbob, role_i: mtrun,
                  role_r: mtrun, d_com: mtfield, cnt: 0 >
   < mrx; Intruder | e_com: mtecom, sec_key: skmrx, ncs: mtfset,
                     msgs: mtfset, agents: mrx U bob,
                     role_i: mtrun, role_r: mtrun, d_com: mtfield,
                     cnt: 0 > .
```

`'cf2Agents1Intruder`, `'IntruderRecognizeAttack`, etc are the meta-level representations of the corresponding constants, rule labels, etc. The result is:

```
rewrites:  28694844 in 5666209ms cpu (5707589ms real) (5064 rewrites/second)
result Strategy:
       stop(11,path(path(path(path(path(path(path(path(path(path(path(
       'cf2Agents1Intruder,
       step('BeginRun, 0, CONF1),
       step('IntruderAcceptEveryMessage, 0, CONF2),
       step('IntruderFakeMessage, 0, CONF3),
       step('Message1Rec, 0, CONF4),
       step('IntruderInterceptMessage, 0, CONF5),
       step('IntruderReplayMessage, 1, CONF6),
       step('Message2RecCorrect, 0, CONF7),
       step('IntruderAcceptEveryMessage, 0, CONF8),
       step('IntruderFakeMessage, 4, CONF9),
       step('Message3RecCorrect, 0, CONF10),
       step('IntruderRecognizeAttack, 0, CONF11))))))))))))
```

where $\overline{\mathtt{CONF1}}$-$\overline{\mathtt{CONF10}}$ are the meta-level representations of all intermediate configurations and `CONF11` is

```
< alice; Agent | e_com: (i, n(alice,1), mrx, n(bob,1)), sec_key: skalice,
                 role_i: mtrun, role_r: mtrun, d_com: mtfield, cnt: 1 >
< bob; Agent | e_com: (r, n(bob,1), alice, n(alice,1)), sec_key: skbob,
               role_i: mtrun, role_r: mtrun, d_com: mtfield, cnt: 1 >
< mrx; Intruder | e_com: (i, n(alice,1), bob, n(bob,1)), sec_key: skmrx,
                  role_i: mtrun, role_r: mtrun, d_com: mtfield,
                  ncs: n(alice,1) U n(bob,1),
                  msgs: ped(pk(alice), cat(n(alice,1), n(bob,1))),
                  agents: alice U bob U mrx > .
```