Formal Specification and Analysis of Active Networks and Communication Protocols: The Maude Experience

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

Rewriting logic and the Maude language make possible a new methodology in which formal modeling and analysis can be used from the earliest phases of system design to uncover many errors and inconsistencies, and to reach high assurance for critical components. Our methodology is arranged as a sequence of increasingly stronger methods, including formal modeling, executable specification, model-checking analysis, narrowing, and formal proof, some of which can be used selectively according to the specific needs of each application. The paper reports on a number of experiments and case studies applying this formal methodology to active networks, communication protocols, and security protocols.

1 Introduction

When a new software technology is first developed, nobody has at first a complete understanding of that new technology. The traditional "build first, learn later" approach is too wasteful and, by providing information too late, does not allow any flexibility to optimize the design and implementation effort. It can also lead to serious integration problems, particularly when different teams are developing different subsystems. All those difficulties apply to Active Networks: it is a novel technology being developed by different teams, and the traditional engineering approaches to deal with its architecture and the integration of subsystems are insufficient. More generally, new methods are needed to properly model and design the increasingly more sophisticated communication systems now in the horizon. We propose that much can be gained from a paradigm shift that uses formal modeling systematically and more aggressively from the earliest phases of system design.

The work presented here focuses on applying a novel, wide-spectrum executable formal specification language, called Maude, to Active Networks and Communication Protocols. Our experience indicates that Maude can substantially help in modeling, symbolically simulating, and analyzing the subsystems of Active Networks and other communication systems, in documenting and ensuring consistency of important parts of the architecture, and in formalizing and analyzing important safety-critical aspects. Part of what makes Maude very well suited for these purposes is its flexible wide-spectrum character: it can deal with very early design phases such as architectures and high-level designs, can be used to quickly develop executable prototypes, and can also be used to generate code. There is also a wide range of options on the kind of analyses that can be performed. One can develop formal models of a system very early, can debug formal specifications—which can be partial and incomplete—by executing them, can do more exhaustive model-checking and symbolic simulation analyses, or, for highly critical subsystems, can in fact do full formal verification using our theorem proving tools. Often much can be gained from the "lightweight" application of our methods through executable specification; the full power of our methods can be reserved for those aspects in which it is more critically needed. The first thing to be gained even from the simplest executable specification uses is a precise documentation of a system's design, that can be used as a clear means of communication between teams, and also to find many bugs and inconsistencies very early on.

The Maude interpreter [7] offers also some useful advantages. The first is efficient executability (1.3 million rewrites per second on a Pentium II for some applications) combined with the ability to trace in detail each execution step. 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. Under assumptions of finite-state or of termination it may even be possible to analyze all executions. The reflective capabilities of rewriting logic and Maude [6] 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 from a given initial state. This is very helpful in uncovering security flaws under unforeseen attack scenarios or examining the behavior of newly designed algorithms. Maude executables, its manual, and a collection of examples and papers have been available on the web since January 1999 (`http://maude.csl.sri.com`).

In summary, using Maude to formally specify and analyze communication systems offers the following advantages:

- *Early insertion of the formal method.* In this way, maximum benefit can be obtained, since the design can be corrected very early, before heavy implementation efforts have been spent.

- *Simplicity and intuitive appeal of the formalism.* The formalism involved, namely rewriting logic [29], is very simple and it is very well suited for specifying distributed systems, in which local concurrent transitions can be specified as rewrite rules.

- *Modeling Flexibility.* Instead of building in a fixed model of concurrency, rewriting logic allows great flexibility to specify many such models [29, 32], including both synchronous and asynchronous models of communication and a wide range of concurrent object systems.

- *Executability.* Rewriting logic specifications are executable in a rewriting logic language such as Maude [7]. This means that the formal model of the protocol becomes an *executable prototype*, that can be directly used for simulating, testing and debugging the specification.

- *Formal analysis and proof.* Since the behavior of a communications protocol is highly concurrent and nondeterministic, a particular simulation run only exhibits *one* among many possible behaviors. Therefore, although direct execution can already reveal many errors and inconsistencies, a much greater confidence in the correctness of the design can be gained by formal analysis techniques in which *all* possible behaviors—up to termination, or up to a certain depth, or up to satisfaction of a specific state condition—are analyzed in detail. This can be done in Maude by means of exhaustive execution strategies that achieve a form of "model checking" analysis of the state space. In addition, formal proofs of highly critical properties can also be carried out using Maude's theorem proving tools [4].

1.1 Applying Maude to Active Networks and Communication Protocols

In collaboration with other teams working on active networks and on communication and security protocols we have applied Maude to formally specify and analyze active networks protocols and algorithms, security protocols, composable communication services, and distributed software architectures. Our experience so far is quite encouraging:

- In [14] we report joint work with group led by J.J. García-Luna at the Computer Communications Research Group at University of California Santa Cruz in which we used Maude very early in the design of a new reliable broadcast protocol for ANs. In this work, we have developed precise executable specifications of the new protocol and, by analyzing it through execution and model-checking techniques, we have found many deadlocks and inconsistencies, and have clarified incomplete or unspecified assumptions about its behavior.

- We have also applied Maude to the specification and analysis of cryptographic protocols [10] and have shown how our model-checking techniques can be used to discover attacks.

- The positive experience with security protocols has led to the adoption of Maude by J. Millen and G. Denker as the basis for giving a formal semantics to their new secure protocol specification language CAPSL and as the meta-tool used to endow CAPSL with an execution and formal analysis environment [13].

- The paper [44] reports joint work with with Y. Wang and C. Gunter at the University of Pennsylvania in using Maude to formally specify and analyze a PLAN [22] active network algorithm.

- In [11] we present an executable specification of a general middleware architecture for composable distributed communication services such as fault-tolerance, security, and so on, that can be composed and can be dynamically added to selected subsets of a distributed communications system.

- In[7] (Appendix E) we present a substantial case study showing how Maude can be used to execute very high level software designs, namely architectural descriptions. It focuses on a difficult case, namely, *heterogeneous* architectures illustrated by a command and control example featuring dataflow, message passing, and implicit invocation sub-architectures. Using Maude, each of the different subarchitectures can not only be executed, but they can also be interoperated in the execution of the resulting overall system.

- As part of a project to represent the Wright architecture description language [2] in Maude we have developed a representation of CSP ("Communicating Sequential Processes", process algebra notation) [23] in Maude. This is compatible with existing tools for analyzing CSP specifications, complements them by providing a rich execution environment and the ability to analyze non-finite state specifications, and provides a means of combining CSP specifications with other notations for specifying concurrent systems.

1.2 Formal Methodology

The formal methodology underlying our approach can be summarized by stating that *a small amount of formal methods can go a long way*. Approaches requiring full mathematical verification of a system can be too costly. Proof efforts should be used judiciously and selectively, carefully choosing those properties for which a very high level of assurance is needed. But there are many important benefits that can be gained from "lighter" uses of formal methods, without necessarily requiring a full-blown proof effort.

The general idea is to have a series of *increasingly stronger methods*, to which a system specification is subjected. Only after less costly and "lighter" methods have been used, leading to a better understanding and to important improvements and corrections of the original specification, is it meaningful and worthwhile to invest effort on "heavier" and costlier methods. Our approach is based on the following, increasingly stronger methods:

1. *Formal specification*. This process results in a first *formal model* of the system, in which many ambiguities and hidden assumptions present in an informal specification are clarified.

2. *Execution of the specification*. If the formal specification is executable, it can be used directly for simulation and debugging purposes, leading to increasingly better versions of the specification.

3. *Formal model-checking analysis*. Errors in highly distributed and nondeterministic systems not revealed by a particular execution can be found by more a sophisticated model-checking analysis that considers all behaviors of a system from an initial state, up to some level or condition. In this way, the specification can be substantially hardened, and can even be formally verified if the system is finite-state.

4. *Narrowing analysis*. By using symbolic expressions with logical variables, one can carry out a symbolic model-checking analysis in which all behaviors not from a single initial state, but from the possibly infinite set of states described by a symbolic expression are analyzed. Some of these analyses are already a special type of formal proof.

5. *Formal Proof*. For highly critical properties it is also possible to carry out a formal proof of correctness, which can be assisted by formal tools such as those in Maude's formal environment [4].

Up to now, we have used methods 1–3 in the case studies mentioned in Section 1.1, reaping important benefits from this use. In the future we also plan to use methods 4 and 5 for selected purposes.

1.3 Outline of this Paper

Rewriting logic and Maude, including the formal modeling of distributed objects, are summarized in Section 2. To make our methods and ideas concrete, we discuss in detail a case study applying Maude to security protocols in Section 3. Other active network and communication protocol applications are discussed in Section 4. Section 5 presents some concluding remarks and discusses future developments.

2 Rewriting Logic and Maude Basics

Rewriting logic is a very simple logic in which the state space of a distributed system is formally specified as an algebraic data type by means of an equational specification consisting of a signature of types and operations Σ and a collection of conditional equations E. The *dynamics* of such a distributed system is then specified by rewrite rules of the form

$$t \to t'$$

where t, t' are Σ-terms, that describe the *local, concurrent transitions* possible in the system, namely, when a part of the distributed state fits the pattern t, then it can change to a new local state fitting the pattern t'. Therefore, a *rewrite theory* is a triple (Σ, E, R), with (Σ, E) an equational specification axiomatizing a system's distributed state space, and R a collection of rewrite rules axiomatizing the system's local transitions. Rewriting logic has proved to be a general and flexible semantic framework for specifying a wide variety of models of concurrency [29, 33], and in particular for specifying concurrent object systems [30].

Maude [5] is a multi-paradigm executable specification language based on rewriting logic [29]. Maude integrates an equational style of functional specification with an object-oriented specification style for highly concurrent and nondeterministic object systems. Maude modules are rewrite theories whose basic axioms are rewrite rules. 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 [5], thus allowing their use for system prototyping and the debugging of specifications.

We briefly summarize the syntax of Maude. There are three types of modules:

- *functional* modules, that are equational theories used to specify algebraic data types; they are declared with the syntax `fmod ... endfm`,
- *system* modules, that are rewrite theories specifying concurrent systems; they are declared with the syntax `mod ... endm`, and
- *object-oriented* modules, that provide special syntax to specify concurrent object-oriented systems but are entirely reducible to system modules; they are declared with the syntax `omod ... endom`.

All the above modules can be *parameterized*, and have an *initial model semantics*. That is, they specify the initial models of their corresponding theories, or, in the case of parameterized modules, the free extension of a model of the parameter theory to a model of the body.

Immediately after the module's keyword, the *name* of the module is given. After this, a list of imported submodules can be added. One can also declare *sorts* and *subsorts* and *operators*. 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` and `comm`, stating, for example, that the operator is associative and commutative. Such attributes are then used by the Maude engine to match terms *modulo* the declared axioms.

There are three kinds of logical axioms, 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`. Functional modules can only have equations and memberships. System and object-oriented modules can have any of the three kinds of axioms. The mathematical variables in such axioms are declared with the keywords `var` and `vars`.

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. In object-oriented Maude modules one can declare *classes* and *subclasses*. Each class is declared with the syntax

$$\texttt{class}\ C\ \texttt{|}\ a_1\texttt{:}\ S_1\texttt{,}\ \ldots\texttt{,}\ a_n\texttt{:}\ S_n$$

where C is the name of the class, and for each $a_i : S_i$, a_i is an attribute identifier, and S_i is the sort over which the values of such an attribute identifier must range. Objects of a class are then record-like structures of the form

$$\texttt{<}\ O\texttt{;}\ C\ \texttt{|}\ a_1\texttt{:}\ v_1\texttt{,}\ \ldots\texttt{,}\ a_n\texttt{:}\ v_n\ \texttt{>}$$

with O the name of the object, $v_1, \ldots v_n$ the current values of its attributes, and with v_i of sort S_i for $1 \leq i \leq n$. Objects can interact with each other in a variety of ways, including the sending of messages. A message declaration

$$\texttt{msg}\ m : p_1 \ldots p_n \rightarrow \texttt{Message}$$

defines the name of the message and the sorts of its 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. 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 in Maude. It 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. For instance,

```
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 reception of a message 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 A Cryptographic Protocol Analysis Application

Cryptographic protocols for key exchange and authentication are the basis for network security. These protocols are often described in a message-list style, i.e., the sequence of messages to be exchanged between the communicating agents is defined as a finite transition system. Though many of the security protocols only involve a very restricted number of messages, it is surprisingly difficult to get the protocols correct. Even under the assumption of “perfect cryptography”, i.e., an encrypted message can only be decrypted with the proper key, it has been shown that protocols are vulnerable to message modification attacks [16, 25]. An intruder can overhear or intercept network communications and can manipulate ongoing communications by faked messages. The highly concurrent nature of cryptographic protocols is responsible for the complexity of proving a cryptographic protocol secure with respect to message secrecy or correct with respect to authentication. This suggests that a formal analysis methodology supporting several analysis techniques, such as simulation, model checking and theorem proving, could be of great value for the design of cryptographic protocols.

We have applied Maude to the specification and analysis of several cryptographic protocols. We illustrate our approach with the help of the Needham-Schroeder Public-Key (NSPK) protocol [34]. In Section 3.1 we illustrate the naturalness of an object-oriented NSPK specification in Maude. The level of specification is detailed enough to capture essential operational aspects and to 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. This translates into additional naturalness and ease of representation.

In order to successfully detect protocol vulnerabilities it is necessary to build not only an abstract model of the protocol but also formal models of the malicious environment, and to define situations in which security goals are violated. It is the combination of the various models of the protocol, the attacker, and the attack that is amenable to formal analysis techniques. In Section 3.2 we describe our formalization of the malicious environment. A breadth-first search strategy and the attack are presented in Section 3.3.

The Maude interpreter allows to efficiently execute the protocol specification. This way we can detect inconsistencies and flaws. But this is not enough to examine all possible behaviors. 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. We explain our strategies in more detail in Section 3.3.

3.1 Specifying and Executing the NSPK Protocol

The NSPK Protocol [34] 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 [25] is:

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

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 that he is 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.

We make this implicit knowledge explicit and clarify fuzzy parts of the description by formalizing the protocol in rewriting logic. In addition to the protocol requirements given above, we provide a Maude specification which allows multiple sessions for an agent. Therefore, an agent may simultaneously participate in several runs of the protocol, and 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 Nonce -> 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 in an inclusion hierarchy by defining subsort relationships. For instance, nonces, principals, and keys are subsorts of `FIELD`. The predicate `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 the empty constant of 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 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 argument of a run is 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 (`Run`), 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.

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

```
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.

```
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 .

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))) .
```

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].

```
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) .
```

[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.

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 the nonce is not in use in a responder session of agent `B`. 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, e_com: C
               role_i: RI U (NI,P,mtfield) >
  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 [5]. 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 responder session with `Bob` as initiator without recognizing that there is another run in which the nonce `NI` is already used. This error clarifies a fact which was not clear from the informal specification: in this protocol nonces play the role of session identifiers. In order to reflect this in the specification we have to change the condition for rule `Message1Rec`. It needs to be checked whether the first nonce in the message is already used in another session (`not(first(F) in RI)`). This example illustrates the usefulness of executing specifications in the Maude engine for validation purposes.

3.2 The Intruder

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 the attribute `ncs`. Similarly, he remembers all seen agent identities in `agents` and all seen messages in `msgs`. 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 rules specifying 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 a key he has, then 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 .
```

3.3 Finding an Attack: a Breadth-First-Search Strategy

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 [25]. This attack is newer and more subtle than the one proposed by Denning and Sacco in [16] which allows an intruder to replay old, compromised public keys. The attacker takes part in two sessions and exchanges information between them ($I(A)$ stands for the intruder masquerading as agent A).

Message 1	$A \rightarrow I : \{N_a, A\}_{PK(I)}$
Message 1'	$I(A) \rightarrow B : \{N_a, A\}_{PK(B)}$
Message 2'	$B \rightarrow A : \{N_a.N_b\}_{PK(A)}$ /∗ *intercepted* ∗/
Message 2	$I \rightarrow A : \{N_a.N_b\}_{PK(A)}$
Message 3	$A \rightarrow I : \{N_b\}_{PK(I)}$
Message 3'	$I(A) \rightarrow B : \{N_b\}_{PK(B)}$

At the end of this attack B believes that he talks to A, i.e., the authentication of the initiator fails.

We can take advantage of the Maude engine to analyze the enriched specification "NSPK+Intruder". For this purpose we need to specify an appropriate analyzing strategy. Thanks to the reflective nature of rewriting logic, 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 5).

For the "NSPK+Intruder" specification we implemented a breadth-first-search strategy (brFS) which searches for possible attacks. To efficiently implement this strategy, we have chosen to use a bounded depth-first-search strategy (bDFS) 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 brFS :
   Module Term QidList QidSet MachineInt
    -> Strategy .
op bDFS :
   Module Term QidList QidSet MachineInt
   -> Strategy .
eq brFS(M,T,Labels,StopLabels,StartDepth)
 = if bDFS(M,T,Labels,StopLabels,StartDepth)
        == stop(0, emptyPath)
   then bDFS(
         M,T,Labels,StopLabels,StartDepth+1)
   else bDFS(
         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 rewrite labels in the specification), a set of rewrite labels which define attack situations and 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.

For NSPK we defined one stop rule: a 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 .
```

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 rewrite 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 rewrite 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 set of stop rules is 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. Running this strategy delivers the attack given in [25].

Notice that our protocol specification is an infinite state specification. We made no restrictions to fit the specification into a finite state tool. This is one of the strengths of the Maude approach. Moreover, the specification is amenable to further forms of analysis such as narrowing analysis and theorem proving. The NSPK example has been chosen for illustrative purposes to emphasize the wide-spectrum modeling and analysis technique of rewriting logic and Maude.

4 Other Applications

4.1 A Reliable Broadcast Protocol

In [14] we report on an ongoing case study in which a new reliable broadcasting protocol (RBP) [20] currently under development at the University of California at Santa Cruz (UCSC) has been formally specified and analyzed, leading to many corrections and improvements to the original design. Traditionally, the only means to document such protocol designs is pseudo-code. Since this is not suitable for execution or formal analysis, it is only in a post-facto way, after the protocol has been implemented, that flaws and inconsistencies are usually discovered. The process of formally specifying the protocol, and of symbolically executing and formally analyzing the resulting specification, has revealed many bugs and inconsistencies very early in the design process, before the protocol was implemented.

RBP performs reliable broadcasting of information in networks with dynamic topology. Reliable broadcasting is not trivial when the topology of the network can change due to failure and mobility. The aim is to ensure that all nodes that satisfy certain connectedness criteria receive the information within a finite time, and that the source is notified about it. The protocol should furthermore incur as low latency and as few messages as possible.

The full Maude specification tackles the cases of sending and receiving messages and acknowledgments, as well as link deletion and link addition, and can be found in [15]. A major advantage of rewriting logic specifications is that they can be validated immediately by executing test cases to provide quick feedback on the specification. This prototyping possibility comes for free. We used this feature every time the specification was modified, and often encountered errors quite easily on quite simple test examples (such as a network of three nodes). In this validation effort, we executed the test cases using Maude’s default interpreter, which simulated an arbitrary run of the protocol for a given initial state of a network. The validation effort helped eliminate errors of syntax and of thought; furthermore, the built-in Maude facilities for tracing an execution were useful for discovering where the error occurred. This validation and correction cycle led to substantial improvements on, and a clear formalization of, the basic ideas of the starting informal protocol.

To substantially increase our confidence in the specification before any costly attempt at a formal proof of correctness, the specification was subjected to close formal analysis using the meta-programming features of Maude to explore all states and behaviors that can nondeterministically be reached from an initial state. Since the specification should be terminating, one could apply a strategy that explores all possible rewrite paths from some given initial state. In particular, we wrote a strategy for finding every non-rewritable state reachable from the initial state. For non-terminating systems, this setting can be modified to give, e.g., every state which is reachable in less than 50 one-step rewrites from some initial state.

We experimented with different, increasingly complex, versions of the protocol. For example, executing the specification using an exhaustive strategy on a clique with three nodes did not produce the hoped-for result, namely a singleton set of irreducible states. Instead, the set of irreducible configurations reachable from the clique of three nodes included a term which indicated a deadlock before the expected end of one round of the protocol. A simple analysis of the output explained the error.

Using Maude's meta-programming features, the user may himself define the rewrite strategies, thereby analyzing the specification in various ways. Although we only needed the quite straightforward exploratory analysis to invalidate our first version, this capability is very useful for analyzing various executions and extracting the relevant information from these automatically. Using the information from the exploratory analysis, the group came up with a new improved version of the protocol.

The effort of formally specifying RBP for dynamic networks using Maude brought to light several weaknesses of the original pseudocode [20]. However, the analysis is not yet finished. For the moment, the following problems were identified and solved (for details see [15]):

1. As described above we could eliminate a deadlock situation in the given protocol.
2. The pseudocode on which we based our first specification of the dynamic RBP was incomplete. In several places it was not clear what are the assumptions about node attributes.
3. A description of initial state was missing, and so was the termination condition for the protocol.
4. Moreover, the original pseudocode was incomplete with respect to how attributes are updated.
5. Some cases were left out in the original specification.
6. Other essential errors were detected using the strategies. For example, we tested a scenario with three nodes a, b, and c where b has the neighbors a and c, a is a source nodes which sends a message and the link between a and b breaks down. Running the protocol with this initial configuration using an exhaustive search strategy delivers three different states of which one is a correct state, the second one reveals an undesired behavior and the third one showed an error in the original pseudocode which has been corrected in the current version.

4.2 Cryptographic Protocol Analysis

As mentioned earlier in this paper, we applied Maude to the specification and analysis of cryptographic protocols and showed how our model-checking techniques can be used to discover attacks.

The TIPE DARPA project focuses on the development of tools for cryptographic protocol analysis. The approach taken is to provide a single common protocol specification language that could be used as the input format for any formal analysis. CAPSL (Common Authentication Protocol Specification Language) [12] has been designed as such a language. It is a modern, easy-to-use specification language for the early specification phase. The main syntactic feature of CAPSL is a message-list style protocol description as can be found in most textbooks published. It is translated to CIL [13], an intermediate language, that expresses the protocol as transition rules in a term rewriting formalism similar to the formalism we used in our experiments with NSPK. The CAPSL Intermediate Language (CIL) serves two purposes: to help define the semantics of CAPSL, and to act as an interface through which protocols specified in CAPSL can be analyzed using a variety of tools. TIPE aims at designing and implementing an integrated protocol environment for the specification and analysis of cryptographic protocols.

Maude has been used in the TIPE project for several purposes. First, the translation from CAPSL to CIL has been carried out in Maude. This way, the formal semantics of CAPSL via its translation to CIL is precisely specified. In the current translator a pre-compiled and type-checked abstract syntax tree is passed to the Maude implementation which determines the corresponding CIL specification. The translator transforms the message list into a set of multiset rewrite rules.

Second, Maude is used as a model-checking tool in the integrated protocol environment and tool-kit. The CIL output of a cryptographic protocol is translated into an executable Maude specification. A metalevel search strategy imports the Maude protocol specification and provides the user with a predefined breadth-first strategy. Search scenarios are automatically derived from the corresponding CAPSL environment specification. The translation from CIL to executable Maude specifications is performed in Maude itself.

4.3 Specifying and Analyzing a PLAN Algorithm

PLAN [22] is a language to program active networks developed at the University of Pennsylvania. In collaboration with Yaw Wang and Carl Gunter at Penn, we have used Maude to formally specify and analyze a PLAN active network algorithm in which active packets scout the nodes of an active network from a source to a destination to find an optimal route relative to a given metric [44]. One interesting aspect of this case study is that it is not clear how to apply traditional network models, such as those used in model checkers, to specify active networks in which active packets carry code that is executed in different nodes and can modify the network. In this regard, the great modeling flexibility of Maude has allowed us to specify the PLAN algorithm in a very direct and simple way that is furthermore executable.

The algorithm has three phases. When a user wants to send a message from a source to a destination, "scout" packets are first sent to try to find an optimal route according to a predefined metric. Then, when a route is found, a flow packet is sent to set up the route. Finally, the desired data packets are sent. Each of these phases is formalized in Maude by appropriate rewrite rules, so that the entire algorithm is specified as a rewrite theory. Since packets and nodes only have local information, it is possible for some data packets to arrive at the destination while some scout packets are still in transit. Also, the route may change while data packets are in transit. Therefore, it is not obvious that the algorithm delivers messages correctly.

The main limitation of testing the algorithm by executing it is that only some behaviors among many are thus explored, while errors may still be lurking in unexplored behaviors. Since the algorithm always terminates, using Maude it is possible to explore all behaviors from a given initial state to check their correctness. This can be done by specifying a suitable strategy at the metalevel. Such a strategy takes the Maude formal specification of the PLAN algorithm and an initial state of the network as inputs, and then explores all the possible behaviors until termination, checking the correct delivery of the messages.

By performing such an analysis we have gained greater confidence of the correct behavior of the algorithm. Of course, even greater confidence could be gained by a narrowing analysis, beginning with a symbolically described family of initial states, or with formal proofs. However, our methodology suggests using those heavier methods only after other errors have already been eliminated by a model checking analysis of the kind described above. Another promising direction would be using Maude to give a formal operational semantics to PLAN. Then, one could use such a formal semantics to analyze and prove properties of PLAN algorithms, that would automatically become formal objects within Maude's logic.

4.4 Composable Distributed Services

In practice NSPK and other protocols provide services designed to ensure desired communication semantics or other properties of the runtime infrastructure. These protocols are not—as it might appear from the usual discussions—the main objectives of the agents involved. It is important to be able to specify and implement services and protocols in a *modular* way, so that they can be installed dynamically and can be composed in many ways to meet complex and changing requirements in open distributed systems. It is also 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?

Composition of system properties is in general a nontrivial matter (c.f. [36, 38, 39]) and a modular approach seems particularly promising in this regard. Formally we want operations that compose two or more protocols to form a new protocol, and that 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 $\mathcal{C}$. Two key questions are: if $\mathcal{S}$ ensures property P for $\mathcal{C}$ under what conditions does $\mathcal{S}$ ensure property P for $\mathcal{S}'[\mathcal{C}]$ where $\mathcal{S}'$ is another protocol; 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 $\mathcal{S}'[\mathcal{C}]$.

Subclassing is one way to obtain some degree of modularity. An alternative is to use reflection to achieve a highly expressive, modular, dynamic formalism. In particular, reflection can be used as a mechanism for installing protocols by composing a protocol specified as a metalevel 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 [1, 3] 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 by adding new metalevels; the *two level actor model* [41] 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. Services are composed by concurrent operation of service meta-actors and interaction via message passing. This model has been used to specify and reason about services such as migration and distributed garbage collection [43], and QoS based admission control in a multi-media system [42] .

Such reflective models can be naturally represented in Maude. In [11] we present an executable specification of a general middleware architecture for composable distributed communication services based on the onion skin model. We also explain how the compositionality of the resulting middleware can be exploited in proving formal properties about the composed system consisting of the middleware services and given applications. The formal model is based on the rewriting logic axiomatization of concurrent objects specialized to distributed asynchronous objects in the spirit of the actor model. It provides a representation of meta objects and an encapsulation of layers of meta-objects in towers. We show how a system of concurrent objects communicating through asynchronous message passing can be naturally transformed into a behaviorally equivalent system of *meta-object towers of height one*. Composition of meta-objects is very simply expressed by stacking meta-objects on top of each other, with the application meta-object at the bottom of the stack. The behavior of meta-object compositions is then formalized by general rewrite rules governing the communication between the different levels of a stack of meta-objects, as well as by the specific additional rules that govern the behavior of meta-objects providing a particular service. Using the initial semantics of Maude it is now easy to formalize notions such as $\mathcal{S}$ ensures property P for $\mathcal{C}$ (written $\mathcal{S}[\mathcal{C}] \models_{\mathrm{ind}} P$). We show how the formal properties expressing different service guarantees—relative to different hostile environments—can themselves also be stated in a parametric way, so that one can combine in a systematic way the properties of different services when composing them, in order to study whether they hold or not for a given composition of services. We show how to model composable communication services as distributed meta-objects and illustrate the basic ideas with a case study treating fault-tolerance, encryption and authentication services and their composition. Formal specification of service guarantees and combination of guarantees for compositions of these services are also discussed.

4.5 Software Architecture Interoperation

To illustrate the flexibility and ease with which different semantic models can be specified and interoperated in rewriting logic we have carried out a substantial case study as part of the DARPA EDCS initiative. The study, entitled “A Software Architecture Interoperation Example,” is documented as Appendix E of [7]. In this study we show how Maude can be used to execute very high level software designs, namely architectural descriptions. It focuses on a difficult case, namely, *heterogeneous* architectures illustrated by a command and control example featuring dataflow, message passing, and implicit invocation sub-architectures. This is accomplished by first defining a general model for *objects* that can interact with each other in a variety of synchronous and asynchronous ways. Then we define a general *dataflow* model as a semantic model for a pipes and filters interaction model as well as a semantic model for *event-based implicit invocation*. We then give an executable high-level specification of a command and control system in which several of these models are combined and interoperated.

To demonstrate the use of Maude to give formal executable and analyzable semantics to software engineering notations, a subset of the Wright architecture description language [2] has been implemented in Maude. This provides an execution environment for Wright specifications as well as the basis for defining additional analysis tools using the reflective capabilities of Maude.

As part of the mapping of Wright to Maude, a mapping of CSP to Maude has been designed and implemented. This mapping is based on concurrent distributed objects (actors), providing as a side benefit a truly concurrent semantics for CSP. The specification of CSP and its machine readable syntax as given in [37] was followed in order to have an independently useful tool for CSP that can interoperate with existing CSP tools and to support interoperation with other notations formalized in Maude such as the CAPSL language for specification of security protocols. This representation of CSP in Maude also provides the starting point for interoperation of CSP specifications and those based on other communication models such as the asynchronous communication of actors, also implemented in Maude.

5 Conclusions and Future Developments

We have presented our experience so far in applying a new formal methodology based on rewriting logic and supported by the Maude language to active networks, communication protocols, and security protocols. The case studies are encouraging and suggest that this methodology, when inserted early in the design process, can uncover many design errors and inconsistencies, can yield unambiguous and executable formal models of the systems being designed, and can be used to reach high levels of assurance about communication systems through a combined series of increasingly stronger techniques.

The case studies discussed in the paper illustrate the use of the techniques 1–3 in the formal methodology of Section 1.2. In the near future we plan to explore more systematically the application of techniques 4–5, namely, the use of narrowing analysis, and of formal proof based on a suitable combination of rewriting and temporal logic. We discuss in more detail each of these techniques in what follows.

5.1 Narrowing

Our systematic search for an attack using a breadth-first strategy and several applications discussed in Section 4 have 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* [19] 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 [26, 27, 28]. 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.

5.2 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 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.

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 [29], 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 [24] for reasoning about object-oriented Maude specifications, and the object-oriented Distributed Temporal Logic (DTL) of Ehrich et al. [18, 9], and the temporal logic of Duarte [17] used to axiomatize the actor model. 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 [8]. 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 Paulson's inductive method [35], Gray and McLean's approach [21], which uses Lamport's TLA notation to specify cryptographic protocols and correctness requirements, and Syverson and Meadows' approach [40], which proposes a logical language for specifying protocol requirements and translates them by hand into the Prolog language of the NRL Analyzer [26, 27, 28]. 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 active networks and 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

This work has been partially supported by DARPA through Rome Laboratories Contract F30602-97-C-0312, by DARPA and NASA through Contract NAS2-98073, by ARPA/AirForce under grant F30602-96-1-0300, by NSF under grant CRR-9633419, and by Office of Naval Research Contracts N00014-96-C-0114 and N00012-99-C-0198.

We thank our fellow members in the Maude team, particularly Peter Ölveczky, Francisco Durán, Narciso Martí-Oliet, Patrick Lincoln, Steven Eker, and Manuel Clavel for their contributions to different aspects of the ideas presented here. We have also benefited from discussions on temporal logic matters with Narciso Martí-Oliet, Hans-Dieter Ehrich, José Fiadeiro, Tom Maibaum, and Carlos Duarte and from many discussions with members of the DARPA Active Networks program, particularly with Kirstie Bellman and Chris Landauer. Finally, we wish to thank Jon Millen at SRI, José García-Luna, Jyoti Rajy, and Brad Smith at UCSC, and Carl Gunter and Yao Wang at UPenn for their important contributions to several of the case studies discussed in this paper.

References

[1] 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.

[2] R. Allen and D. Garlan. Formalizing architectural connection. In *Proceedings 16th International Conference on Software Engineering*, 1994.

[3] M. Astley. *Customization and Composition of Distributed Objects: Policy Management in Distributed Software Architectures*. PhD thesis, University of Illinois, Urbana-Champaign, 1999.

[4] M. Clavel, F. Dur´an, S. Eker, and J. Meseguer. Building equational proving tools by reflection in rewriting logic. In *Proc. of the CafeOBJ Symposium '98, Numazu, Japan*. CafeOBJ Project, April 1998. `http://maude.csl.sri.com`.

[5] M. Clavel, S. Eker, P. Lincoln, and J. Meseguer. Principles of Maude. In Meseguer [31], pages 65–89.

[6] M. Clavel and J. Meseguer. Reflection and Strategies in Rewriting Logic. In Meseguer [31], pages 125–147.

[7] Clavel, M. and Dur´an, F. and Eker, S. and Lincoln, P. and Martí-Oliet, N. and Meseguer, J. and Quesada, J. *Maude: Specification and Programming in Rewriting Logic*. SRI International, Computer Science Laboratory, Menlo Park, CA, January 1999. `http://maude.csl.sri.com/manual/`.

[8] G. Denker. From Rewrite Theories to Temporal Logic Theories. In H. Kirchner and C. Kirchner, editors, *2nd Int. Workshop on Rewriting Logic and Its Applications (WRLA'98), Pont-A-Mousson, France, September 1-4, 1998*. Elsevier Science B.V., Volume 15 of Electronic Notes in Theoretical Computer Science, `http://www.elsevier.nl/locate/entcs/volume15.html`, 1998.

[9] 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.

[10] G. Denker, J. Meseguer, and C. Talcott. Protocol Specification and Analysis in Maude. In N. Heintze and J. Wing, editors, *Proc. of Workshop on Formal Methods and Security Protocols, 25 June 1998, Indianapolis, Indiana*, 1998. `http://www.cs.bell-labs.com/who/nch/fmsp/index.html`.

[11] G. Denker, J. Meseguer, and C. Talcott. Rewriting Semantics of Distributed Meta Objects and Composable Communication Services, 1999. working draft.

[12] G. Denker and J. Millen. CAPSL and CIL Language Design: A Common Authentication Protocol Specification Language and Its Intermediate Language. CSL Report SRI-CSL-99-02, Computer Science Laboratory, SRI International, Menlo Park, CA 94025, 1999. `http://www.csl.sri.com/~denker/pub_99.html`.

[13] G. Denker and J. Millen. CAPSL Intermediate Language. In N. Heintze and E. Clarke, editors, *Workshop on Formal Methods and Security Protocols (FMSP'99), July 5, 1999, Trento, Italy (part of FLOC'99)*, 1999. `http://cm.bell-labs.com/cm/cs/who/nch/fmsp99/`.

[14] Denker, G. and Garcia-Luna-Aceves, J.J. and Meseguer, J. and Ölveczky, P. and Raju, J. and Smith, B. and Talcott, C. Specification and Analysis of a Reliable Broadcasting Protocol in Maude. In B. Hajek and R. Sreenivas, editors, *Proc. 37th Allerton Conference on Communication, Control and Computation*, 1999. url-http://www.comm.csl.uiuc.edu/allerton.

[15] Denker, G. and Garcia-Luna-Aceves, J.J. and Meseguer, J. and Ölveczky, P. and Raju, J. and Smith, B. and Talcott, C. Specifying a Reliable Broadcasting Protocol in Maude. Internal report, Computer Science Laboratory, SRI International, Menlo Park, CA 94025, 1999. `http://www.csl.sri.com/~denker/pub_99.html`.

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

[17] C. H. C. Duarte. *Proof-theoretic Foundations for the Design of Extensible Software Systems*. PhD thesis, Imperial College, University of London, 1999.

[18] 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.

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

[20] J. J. Garc´ı a-Luna. Reliable Broadcasing in Computer Networks. Manuscript; University of California at Santa Cruz, Computer Science Department, January 1998.

[21] 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*, pages 108–116. IEEE Computer Society Press, 1995.

[22] M. Hicks, P. Kakkar, J. T. Moore, C. A. Gunter, and S. Nettles. PLAN: A Packet Language for Active Networks. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming Languages*, pages 86–93. ACM, 1998.

[23] C. Hoare. *Communicating Sequential Processes*. Prentice Hall International, 1985.

[24] U. Lechner and C. Lengauer. Modal–μ–Maude —Specifi cation and Properties of Concurrent Objects. In B. Freitag and C.B. Jones and C. Lengauer and H.-J. Schek, editor, *Object Orientation with Parallelism and Persistence*, pages 23–46. Kluwer, 1996.

[25] G. Lowe. Breaking and fi xing the Needham-Schroeder public-key protocol using FDR. In *Proceedings of TACAS*, volume 1055 of *Lecture Notes in Computer Science*, pages 147–166. Springer-Verlag, 1996.

[26] C. A. Meadows. Formal Verifi cation of Cryptographic Protocols: A Survey. In J. Pieprzyk and R. Savafi -Naini, editors, *Advances in Cryptology - Asiacrypt '94*, pages 133–150. Springer, 1995. LNCS 917.

[27] C. A. Meadows. Analyzing the Needham-Schroeder Public Key Protocol: A Comparison of Two Approaches. In E. Bertino, H. Kurth, G. Martella, and E. Montolivo, editors, *Proc. of European Symposium on Research in Computer Security (ESORICS'96)*, pages 351–364. Springer, 1996. LNCS 1146.

[28] C. A. Meadows. Language Generation and Verifi cation in the NRL Protocol Analyzer. In *Proc. 9th Computer Security Foundations Workshop*, pages 48–61. IEEE Computer Society Press, June 1996.

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

[30] 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.

[31] J. Meseguer, editor. *Rewriting Logic and Its Applications, First International Workshop, Asilomar Conference Center, Pacifi c Grove, CA, September 3-6, 1996*. Elsevier Science B.V., Electronic Notes in Theoretical Computer Science, Volume 4, `http://www.elsevier.nl/locate/entcs/volume4.html`, 1996.

[32] J. Meseguer. Rewriting logic as a semantic framework for concurrency: a progress report. In *Proc. CONCUR'96, Pisa, August 1996*, pages 331–372. Springer LNCS 1119, 1996.

[33] 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.

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

[35] L. Paulson. The inductive approach to verifying cryptographic protocols. *Journal of Computer Security*, 6(1):85–128, 1998.

[36] J. Peleska. On a unifi ed 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.

[37] W. Roscoe. *Theory and Practice of Concurrency*. Prentice Hall, 1998.

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

[39] J. 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`.

[40] 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.

[41] N. Venkatasubramanian. *Resource Management in Open Distributed Systems with Applications to Multimedia*. PhD thesis, University of Illinois, Urbana-Champaign, 1998.

[42] N. Venkatasubramanian, G. Agha, and C. L. Talcott. Specifying composable services for qos-based distributed resource management, 1998. submitted.

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

[44] B.-Y. Wang, J. Meseguer, and C. A. Gunter. Specifi cation and formal analysis of a PLAN algorithm in Maude. Manuscript, September 1999.