MuCAPSL *

Jon Millen and Grit Denker
Computer Science Laboratory
SRI International
Menlo Park, CA 94025, USA
{millen | denker}@csl.sri.com

Abstract

Secure group communication protocols have been designed to meet needs such as secure management of group membership, confidential group communication, and access control. New languages and models are necessary to appropriately capture the concepts of such protocols and make them amenable to formal analysis.

For this purpose, we developed MuCAPSL (Multicast Common Authentication Protocol Specification Language). In this paper we introduce the MuCAPSL features and motivate our design decisions by illustrating the practical use of MuCAPSL with the help of a simplified version of the secure group communication protocol used in SecureSpread. We also briefly introduce MuCAPSL's intermediate language MuCIL which serves as an interface language for analysis tools.

1 Introduction

A variety of new protocols and frameworks have been designed to create multicast groups on a network and support secure group communication, such as GDOI [3] and GSAKMP [11]. Key exchange protocols for secure communication have been extended to the group setting. Some of these are extensions of the symmetric key distribution techniques used in virtual private networks, as in Enclaves [10], others are extensions of Diffie-Hellman, such as Group Diffie-Hellman (GDH) [21, 22] and its authenticated form A-GDH [2]. Some schemes are hierarchical, either physically as in Iolus [18], or abstractly as in various key graph approaches [23, 12]. Different key management approaches can be offered as options over basic reliable multicast frameworks, as in Secure Spread [1].

There have been only a few results on the formal analysis of group management protocols; Pereira and Quisquater analyzed A-GDH [20], and Meadows discovered security flaws in early versions of GDOI [14]. The analysis of group management protocols poses new challenges for formal analysis techniques. New language concepts are necessary to appropriately capture the features of such protocols. Moreover, analysis techniques and tools have to be revised and extended.

MuCAPSL (Multicast Common Authentication Protocol Specification Language) has been designed to meet these needs. MuCAPSL is a high-level, yet mathematically well-founded protocol specification language that allows easy transformation of published descriptions of secure group communication protocols into an intermediate language suitable as an interface to many formal analysis techniques and tools. The ideas behind MuCAPSL and its special features for facilitating group multicast protocol specification and a brief summary of the term-rewriting model underlying the intermediate language are described in this paper.

This paper is primarily about the design of MuCAPSL. We have begun some work on the analysis of multicast protocols using our semantic model, summarized briefly in Section 5.2. More details on the intermediate language and its application for protocol analysis are presented in [16, 8]. Background on the project and the parent unicast language CAPSL may be found in [7].

2 Review of CAPSL

MuCAPSL is based on concepts that appeared in the CAPSL language for unicast protocols. CAPSL

* This work was supported by DARPA through SPAWAR Systems Center under Contract N66001-00-C-8014.

philosophy is retained where it still applies, so to understand MuCAPSL it helps to review its foundation in CAPSL.

CAPSL was designed to formalize the kind of protocol specifications that appear in textbooks and articles. In those sources, a protocol is given as a sequence of messages in what has been called "Alice and Bob" style. A typical message exchange might appear as:

$$A \to B : A, \{N_A, B\}_{K_B}$$
$$B \to A : B, N_A$$

The accompanying text would describe the types and purposes of the variables.

A CAPSL version declares variables as in a strongly-typed programming language, but there are some additional characteristics of CAPSL that are not usually found in programming languages. Consider the CAPSL specification of the example above:

```
PROTOCOL Example;
VARIABLES
  A, B: PKUser;
  Na: Nonce,FRESH,CRYPTO;
ASSUMPTIONS
  HOLDS A: B;
MESSAGES
  A -> B: A,{Na,B}pk(B);
  B -> A: B,Na;
GOALS
  PRECEDES B: A| Na;
END;
```

Certain features of the above specification are significant and are retained in MuCAPSL:

- Variables representing parties in a protocol are usually of some specialized datatype for which protocol-related functions are defined. In this case, a party B of type PKUser has a public key $\mathtt{pk}(B)$.

- Variables have *properties* like FRESH and CRYPTO above, representing assumptions relevant to attack analysis that cannot be expressed simply by defining a type for them. FRESH, for example, means that the variable can be given a new value that has never (for purposes of analysis) been used before.

- Attention is paid to implementability considerations, such as ensuring that a variable receives a value somehow before it is used. This is the reason for the "`HOLDS A: B`" assumption.

- Security goals for the protocol may be specified. PRECEDES is a kind of authentication goal. In this protocol, A can establish that there is a B session holding N_a.

Other characteristics of CAPSL and MuCAPSL are not visible on the surface, but are important for understanding the meaning of specifications. The most important of these is the *latch* behavior of protocol variables. Variables may be uninitialized or undefined at the beginning of a protocol session, but once they receive a value, the value is fixed for the remainder of the session. This behavior reflects the protocol designer's view of variables as having a global or shared value, even though they are stored locally. It affects the interpretation of messages, since it implies, for example, that the value of N_A received by A in the second message should be compared with the original, to detect error conditions, rather than be accepted as a new value for it.

The latch behavior of protocol variables affects the semantics of equations. CAPSL allows equational actions like $K = \mathtt{pk}(B)$ to be interspersed with messages. An equational action is usually a comparison test that must be satisfied at that point, otherwise the protocol session aborts. However, if the left side of the equation is a variable that has not yet received a value, the equation is interpreted as an assignment.

In MuCAPSL, there are two kinds of variables: protocol variables as in CAPSL, having the latch property; and *group member attributes*, which do not. This distinction is important to understanding the conceptual design of MuCAPSL. The difference in behavior is related to the other major difference between protocol variables and attributes, namely that protocol variables are transient – they disappear when the protocol run ends – while attributes are persistent – the last value stored in a previous run is available at the beginning of the next run (of either the same or a different protocol) by the same group member.

2.1 Specification Components

A MuCAPSL or CAPSL specification is made up of three kinds of modules: *typespec*, *protocol*, and *environment* specifications, usually in that order.

Abstract data type specifications (called *typespecs*) introduce new data types and define cryptographic operators and other functions axiomatically. Standard typespecs are included automatically and others may be supplied by the user. Our abstract data type approach borrows from those appearing in modern systems like PVS [19] and Maude [5]. Functions are characterized axiomatically, rather than defined procedurally, with methods as in languages like Java. However, as in object-oriented languages, certain MuCAPSL objects – group members – do have modifiable states. Protocol agents also have states, but they are not declared as objects of an explicit type, but, instead, are defined using separate protocol specifications.

There may be more than one protocol specification. In CAPSL, this occurrs when a top-level protocol invokes one or more subprotocols. In MuCAPSL, subprotocol delegation may occur, but the main reason to have multiple protocols is to specify several distinct functions for group management, such as initialize, join, leave, merge, and re-key.

Environment specifications are optional; they are used to set up particular network scenarios for the benefit of search tools. They are not discussed here.

2.2 CAPSL Tools

The purpose of a CAPSL specification is primarily to support vulnerability analysis using formal methods. There are many analysis tools and methods, but none of them is specifically included in the CAPSL environment. Instead, the goal of CAPSL is to support a variety of such tools by defining a universal language they can share.

The principal CAPSL tool is a translator that parses and checks a CAPSL specification, and translates it into a term-rewriting specification of a state transition model. The term-rewriting notation is called CIL. The CIL form of a CAPSL specification expresses its semantics and serves as an intermediate language. It is possible to write translators ("connectors") from CIL to the input language of various analysis tools, and some of those have in fact been written, including one for Athena [15] and one for the Naval Research Laboratory Protocol Analyzer, as well as experimental ones for PVS, the SRI Constraint Solver [17], and Maude [6].

The CIL language has been extended to MuCIL to support multicast group management protocols in MuCAPSL, and there is now a MuCAPSL to MuCIL translator.

3 MuCAPSL Concepts

There are two prominent differences between CAPSL and MuCAPSL. One is that there is a new kind of protocol variable, mentioned above, called a group member attribute. Attributes and a built-in group member type are discussed in Section 3.1. In contrast to protocol variables, attributes can be modified. As a consequence, a new syntactic notation is needed to determine whether received values for an attribute are to be compared with the existing value or to be replaced. This is done in Section 3.3.

The other big difference is that the Alice-and-Bob style of specification is no longer used, even though the bulk of the protocol specification consists of message sequences. The roles of different parties in the protocol are specified separately as presented in Section 3.2.

3.1 The Group Member Type

In CAPSL, messages are sent from and to *principals* – an object of some subtype of Principal, like PKUser. In MuCAPSL, the sender and receiver of a message is an object of some subtype of GroupMember. The GroupMember type has a special status in MuCAPSL as the only kind of object for which attributes may be defined. GroupMember is not a subtype of Principal; rather, a group member is defined with two components, the principal who owns it and the group ID of the group it belongs to. In this way, the same principal may have a presence in two or more groups, as a different group member, holding a different attribute set for each. We sometimes use the term "agent" for "group member" when we wish to de-emphasize the particular group it is associated with, for example when it has just been deleted from the group.

The group ID is meant to persist over different views of the group resulting from membership changes. A new group ID is created only when a totally new group instance is formed from an initial group member or set of members.

We introduce the type GroupMember as the default type for group members in the following specification:

```
TYPESPEC GROUPMEMBER;
TYPE GroupMember: Object;
FUNCTIONS
   owner(GroupMember): Principal;
   groupid(GroupMember): Nat;
END;
```

Note that the owner and groupid are "functions" rather than "attributes." The difference is that attributes can be modified, while if one changes the owner or groupid of a group member it is not the same group member any more. Attributes are discussed in more detail in the next section.

The GroupMember type scheme has a slight inconvenience when group member types are hierarchical - one is a subtype of another - and it is desired to make the owner of one of the lower types a special kind of principal, like PKUser, when its parent owner is only a kind of Principal. MuCAPSL, like CAPSL, does not support multiple inheritance. However, it is still possible to override lower-level function signatures in subtypes to make them more specific. So, although the owner of a GroupMember has type Principal, one can declare a kind (subtype) of GroupMember for which the owner is a subtype of Principal, like PKUser.

3.1.1 Attributes in MuCAPSL

In the group protocol setting, agents need to store keys or other information over several protocol sessions, and they also need to be able to assign new values to group information. For example, members of a group may be ordered and have a sequence number. When a new member is added with a join protocol, existing members retain their old sequence number. When a member is deleted, other members after it in the sequence will decrement their sequence numbers. Persistent but modifiable information like this is stored in attributes. Attributes are specified in typespecs for group member types in a new ATTRIBUTES section. For example,

```
TYPESPEC FRIEND;
TYPES Friend: GroupMember;
ATTRIBUTES
  groupSize: Nat;
  groupKey: Skey;
END FRIEND;
```

Like protocol variables in CAPSL, the values of attributes can be set and tested explicitly by equational actions and implicitly as a result of receiving messages. Unlike protocol variables, attributes can receive a succession of different values. There are some very interesting syntactical and semantic issues regarding how to handle attribute values in messages and equations; those are discussed later after we have introduced the way roles are specified in MuCAPSL.

Attributes, unlike protocol variables, can be functions, in which case they represent local tables. An attribute declared as `keytable(Principal): Skey` holds a key `keytable`(A) for any principal A, and the key for A can be updated within a protocol run.

Attributes can also be arrays, which also represent local tables. Arrays, unlike functions, are objects of a data type that can be sent in messages, and they can be protocol variable values as well. Messages containing arrays are quite useful for specifying some group management protocols.

3.2 Role Separation

The second overwhelming difference between CAPSL and MuCAPSL is the separation of roles in the specification. A CAPSL protocol specification has a single MESSAGES section, in which the list of messages indicates all of the different roles played by participants in the protocol: initiator, responder, key server, etc., typically associated with principal names such as A, B, S, etc.

In a group multicast protocol, there is a variable collection of members $M_1, ..., M_n$, but one can still identify a small fixed number of roles, in the sense of distinct behavioral sequences. In the GDH.2 key distribution protocol [21], for example, there are three roles (see Figure 1): the initiator M_1, the "middle" role played by $M_2, ..., M_{n-1}$, and the final role played by M_n. The middle members $M_2, ..., M_{n-1}$ all receive and send the same kinds of messages, and perform the same computations. Their common role is named "M_i" in the figure.

If one attempts to connect the message arrows in the figure into a single message sequence, one finds quickly that there are more possibilities than are immediately apparent or easy to reconcile. There should be a loop from M_i to itself, and one might forget to connect M_1 directly to M_n in case $n = 2$. And what if the group only has one member? We avoid these problems in MuCAPSL by specifying each role separately. Conditions for instantiating the roles and determining message destinations are given in detail as part of the specification.

Role separation has syntactic consequences. For example, even though the two-message protocol in Secion 2 is not a group multicast protocol, we can still express it in MuCAPSL. The MuCAPSL version looks like this:

```
SUITE EXAMPLE;

TYPESPEC PKGM;
TYPES PKGroupMem: GroupMember;
FUNCTIONS
```

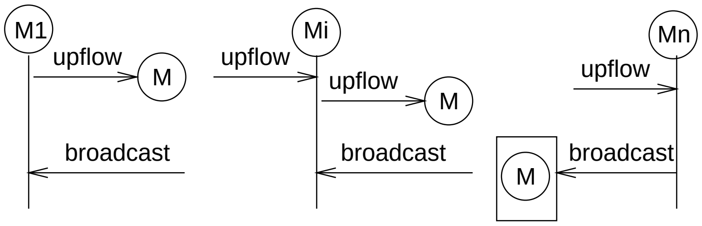

Figure 1. GDH.2 Key Distribution Protocol

```
  owner(PKGroupMem): PKUser;
END;

PROTOCOL Example;
VARIABLES
  Na: Nonce,FRESH,CRYPTO;
  A,B: PKUser;

ROLE Init: PKGroupMem;
ASSUMPTIONS
  HOLDS A,B;
  A=owner(Init);
MESSAGES
 -> B: A, {Na,B}pk(B);
 <-: B,Na;
END;

ROLE Resp: PKGroupMem;
ASSUMPTIONS
  HOLDS B;
  B=owner(Resp);
MESSAGES
  <-: A, {Na,B}pk(B);
  -> A: B,Na;
END;

GOALS
  PRECEDES Resp: Init | Na;
END Example;
END EXAMPLE;
```

We have introduced two operator symbols, `->_:_` and `<-:_`, for sending and receiving messages, respectively. Since we are specifying a protocol from the viewpoint of each role, the sender in a "`->_:_`" message and the receiver in a "`<-:_`" message is implicitly the owner of the group member playing the role that is being specified. Therefore, we only need to specify the intended receiver in a sent message, and we just omit the receiver if the message is multicast. A sending message without a receiver is understood as a multicast to the group to which the sender belongs.

The origin of a received message is not indicated, because header information is not available to the role process. The claimed identity of the sender can be included in the message as a message field. Note that the role variables are of type GroupMember, which is not a message field type. But the owner( ) is of type Principal and may be included as a message field and used as a destination address.

3.3 Assignment vs. Testing

In CAPSL, when a protocol variable is received in a message or it appears on the left side of an equation action, the CAPSL translator can determine whether the variable is assigned a value or used in a comparison. This is possible because of the latch property: protocol variables are never modified once they initialized. Thus, if a variable has not yet been assigned a value, then the operation is interpreted as an assignment, but if it already has a value, the equation must be a comparison.

Attributes in MuCAPSL may be modified, so an explicit syntactic notation is needed to determine whether a received value is to be compared with the existing value or to replace it. One obvious possibility is to introduce an assignment operator, like ':=', or use '=' for assignment and '==' for comparison, as in C and Java. However, there is another problem we have to solve first, and the solution to that one makes it redundant to have an assignment operator.

Consider receiving a message `<-: X,Y` where X and Y are attributes. What if the intent is to replace X with the value in the message, but to compare Y with the value in the message? We have to distinguish the message fields syntactically. It appears that replacement is more often intended, and we assume that as the default case. When a comparison is intended, we place '?' before an attribute to indicate that. The example would be written as `<-: X,?Y`.

In order to be consistent, we also apply the same rule for protocol variables. Nevertheless, since in the case of protocol variables the decision whether it is an assignment or a test can be made from the context and does not require an explicit mark-up of the variable, the translator will only give a warning if a protocol variable is not marked with a '?' when the variable has been initialized and a test for equality has to take place, or if it is marked but it has not been initialized.

Now we can handle equations the same way. The equation `X = Y` is an assignment into X, and the equation `?X = Y` is a comparison test!

4 A Group-Diffie-Hellman protocol

Some of the new features of MuCAPSL are illustrated in this section with the help of a protocol example based loosely on Secure Spread [1] and its application of a form of group Diffie-Hellman called IKA.2 in [22].

Our protocol GKA-S (for GKA-simplified) handles membership messages for multiple join and leave events in a multicast group. The result of a successful membership message is the installation of a new membership view in which all current group members share a new group key. A ("cascading") membership message specifies the next group view, with the new set of members and a view identifier. Before we discuss in more details the various roles of the protocol, we introduce the required group member types for GKA-S.

4.1 Member Types and Attributes

For GKA-S we specify two types: CLQ_CTX and GKA-S_CTX. CLQ_CTX holds all the information necessary to participate in the key agreement protocol. GKA-S_CTX is a subtype of CLQ_CTX and has further attributes that hold information that needs to be stored for the Secure Spread group communication algorithm. Agents that engage in a key agreement algorithm are identified by a natural number p. Each agent has a nonce N_p and a set of current group members M. It also stores the last partial key token PT, the last final key token FT, and keylist KL that it received. Ks holds the current group key. A GKA-S member also stores the latest membership information such as the view identity and the transitional, merge, and leave sets of the installed view.

```
TYPESPEC CLQ_CTX;
TYPES CLQ_CTX: GroupMember,MUTEX;
CONSTANTS g: Skey;
ATTRIBUTES
  p: Nat;      /* position in
                  group_members_list */
  Np: Skey;    /* Mi's session
                  random number */
  Ks: Skey;    /* current group
                  shared key */
  M: SParray; /* member list */
  KL: Sarray; /* key list */
  PT: Skey;    /* Partial token */
  FT: Skey;    /* Final token */
END;

TYPESPEC GKA-S_CTX;
TYPES GKA-S_CTX: CLQ_CTX;
ATTRIBUTES
  mb_id: Nat;
  /* group view unique id */
  vs_set: SParray;
  /* transitional set */
  merge_set: SParray;
  leave_set: SParray;
  Me: Principal;
  /* assume: initialized to owner */
END;
```

The MUTEX property of CLQ_CTX means that a group member of this type is single-threaded; it supports only one role process at a time, enforced with an invisible semaphore. This comes into play later when we discuss flow of control.

The types Sarray and SParray are subtypes of the general CAPSL array type. An Sarray is an array of Skeys. (Presently, MuCAPSL does not support parametric types, so arrays of different kinds of values must be declared separately.) A Parray is an array of principals; the SParray subtype is an "ordered set" array, in which members are ordered and no member appears twice. Among the operators defined for SParray are *last(S)* to determine the last principal in the ordered set S, *add(P,S)* to add a new principal to a set, *setequal(S,S)* to test two sets for equality, and *size(S)* to determine the size of an ordered set.

4.2 Roles

The GKA-S algorithm, like GKA, could be presented as the state graph in Figure 4.2 (see also [1]). The state abbreviations stand for Secure, CascadingMembership, PartialToken, FactorOut, FinalToken, and KeyList. Except for the Secure state, the states are named according to the type of message they are waiting to receive. Except for the CM message, the messages contain partial key information supplied in Secure Spread through calls to the

Cliques API. In our specification, we need to represent the content of those messages. In the example, for simplicity, we do not include the digital signatures on messages that are present in Secure Spread.

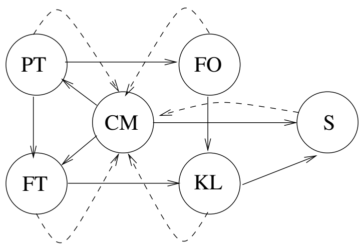

Figure 2. GKA-S

The GKA-S is normally in the Secure state, passing encrypted data messages between the user application and the network. A membership message leads to state transitions necessary to compute a new group key. A new membership message, which may arrive before processing for the previous one has completed, is preceded by a "Flush" message causing a (dashed) transition to the CM state. In Secure Spread, a membership message indicating that some members have left is preceded also by a transitional signal, which is passed upward to the application. Transitional signals are not represented in this example, or any other communication with the application.

When there are multiple transitions from a state, the path taken by a group member depends on its role in the new membership list, one of: Alone, New, Last, Chosen, and Other. Agents that join a group belong to either of the first three roles. The special case in which only one member constitutes the new group is covered by the role "Alone." If several agents join the group, then one among them is identified of being the last agent to join (role "Last") whereas all others behave as specified in a role called "New." Among the transitioning agents (remaining from the prior view) one is chosen to be the group controller ("Chosen") whereas the behavior of all other transitioning agents can be specified in one role "Other."

Agents that engage in GKA-S are given an index p, in our specification, which orders the members such that new ones have the higher indices. Chosen has the highest index of the transitional members, and the one in role Last has the highest index.

The group key is computed from secret random contributions of each group member. For this purpose, each agent has a nonce N_p. The group key K_s is the exponentiation base g raised to the product of all nonces $\Pi_{i=1..n} N_i$ of group members.

Figure 3 illustrates the communication between group members for a membership message in which M_1 and M_2 are previous members of the group, and M_3 and M_4 join the group. M_2 is the chosen group controller and M_4 is the last new member to join the group. The figure assumes that the membership message has already been received. The group controller sends a partial-token message with a new key contribution N_2' to the first joining member M_3. The diagrams indicate how partial token messages are chained upwards, a final-token message is multicast from Last, factor-out messages are sent from all members to Last, and finally Last multicasts a long key-list message from which other members extract one field they can use to compute the final key, using their own nonce.

From studying the message flow, one sees that there is a deterministic sequence of state transitions and message receptions for each role, except for the special flush transitions. For example, whether the next state after PT is FO or FT depends on whether the role is Last or not. This is important for writing MuCAPSL specifications, in which each role normally goes through a single sequence of message sends and receives. Flush handling is accomplished with the help of a separate role that listens for flush requests and interrupts the main processing with the new MuCAPSL "abort" statement.

The Chosen role is specified below for illustration. This specification has examples of array indexing and array operations such as size and last, as well as arithmetic operations, including exponentiation (^).

```
ROLE Chosen: GKA-S_CTX;
ASSUMPTIONS
  Me=owner(Chosen);
MESSAGES
  <-: mb_id,M,vs_set,merge_set,leave_set;
      /* membership msg */
  last(vs_set) = Me;
  size(merge_set)>0;
  p = size(vs_set);
  NEW Np;
  -> merge_set[1]:  PT^Np;
     /* first new PT msg */
  <-: FT;
     /* wait for final token msg */
  -> last(merge_set): Me, FT^(1/Np);
  <-: KL IF size(KL)=size(M)-1;
     /* wait for key list */
  Ks = KL[p]^Np;
```

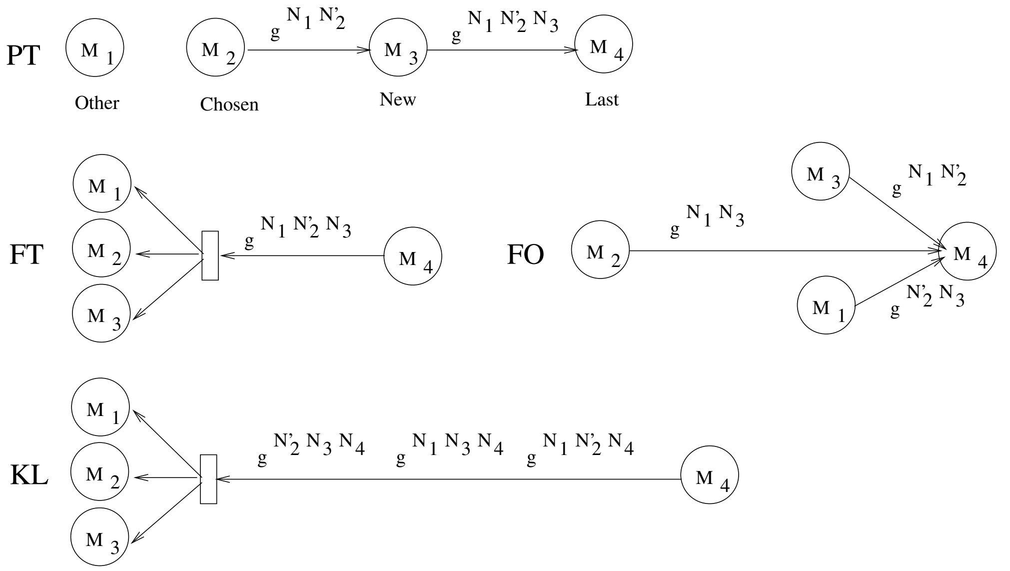

Figure 3. IKA.2 Key Distribution Message Flow

```
END Chosen;
```

Note that the role is confirmed after receiving the membership message, with the tests on the size of merge_set and vs_set. Since other roles start this way, the model is written as though the process nondeterministically selects a role, then aborts it if it discovers that the conditions for that role are not satisfied. This is unrealistic from an implementation point of view, but it generates the same set of possible traces as a system that might crash for no particular reason after receiving a membership message, or fail to receive a membership message, and therefore it yields the same results for security analysis.

4.3 Control Flow: Aborts and Loops

Ideally, MuCAPSL control flow is straight-line for each role. Exceptions are possible and sometimes necessary. One exception occurs in GKA-S to handle flush messages.

```
ROLE Flush: GKA-S_CTX;
MESSAGES
  <-: flush;
  ABORT;
END Flush;
```

This role listens for the flush message ("flush" is just a constant declared at the protocol level) and invokes the ABORT action. The effect of ABORT is to terminate all other processes acting for the same group member. The rewrite-rule semantics of ABORT is simple for a single-threaded group member, and more complex for one that supports multiple concurrent role processes. The single-thread property is indicated by the MUTEX keyword in the GKA-S_CTX typespec earlier. The flush role, or any role that performs an ABORT, runs concurrently. An ABORT resets the implicit mutual exclusion semaphore so that a new role may (nondeterministically) begin and listen for the next CM message.

Another departure from straight-line flow of control occurs when a role must collect a set of messages from other group members in order to compute a key or for other purposes such as determining a majority vote. The Last role in GKA-S collects factor-out messages to assemble the final key-list message. There is a DO-UNTIL construct for this purpose.

```
ROLE Last: LastCTX;
ASSUMPTIONS
  Me=owner(Last);
MESSAGES
  <-: mb_id,M,vs_set,merge_set,leave_set;
  Me = last(merge_set);
  size(M)>1;
```

```
  <-: PT;  /* wait for partial
              token message */
  p=size(M);
  ->: PT;
  FMlist = panull;
  DO      /* wait for factor out
             messages */
    <-: m,FO[pos(m,M)];
    FMlist = add(m,FMlist);
  UNTIL setequal(add(Me,FMlist),M);
  NEW Np;
  Ks = PT^Np;
  KL = FO^^Np;
  -> : KL;
END Last;
```

The Last role specification makes use of three additional attributes that are declared for an extended subtype of GKA-S_CTX called LastCTX. They are: FO, the array of factor-out values; FMlist, the set of principals who have returned the FO message to Last, and a variable m of type Principal used as the loop variable.

Note how the exponentiation with N_p is distributed over all elements of the *FO* array with the elementwise array operator `^^`. This is just infix notation in MuCAPSL for an operation defined in the Sarray typespec. At present, there are several such notational conveniences. Such syntax extensions can be added easily, but still only by the MuCAPSL developers.

In the previous subsection we noted that the nondeterministic selection of roles, confirmed by a test after receiving the CM message, was not natural from an implementation point of view. If a specification is desired that is closer to the implementation state graph, it is possible to make one. The idea is to create a role for each state, and to include an additional attribute, 'State', that determines the role required for the next process. For example, after the CM state-role has examined the CM message and determined whether it should go into a Last role, it can set the State attribute properly to PT or FT and terminate. Then either PT or FT may begin, as appropriate, and so on.

Finally, there is a construct IF-THEN-ELSE for conditional branching within a role. This is convenient when there is a minor choice that does not justify starting a new role.

4.4 Group Security Goals

The fundamental security goal of a group protocol is the same as that for a unicast key distribution protocol: the session key held by each participant should be disclosed only to legitimate participants. In a group management protocol, the statement of this property is more challenging because it is more difficult to specify who the participants are. In some cases, the group membership might not be known at all, and there is merely an assurance that joins are made according to some policy or only by the group leader. Examples of policies particular to group security management are the "forward security" and "backward security" policies, mentioned, among other places, in [12], preventing previous group members and new group members from reading messages sent to new or old views of the group, respectively.

MuCAPSL provides slight extensions of the types of goal statements found in CAPSL, which are very general and are capable of representing group security goals. Goals are stated for protocols. The three kinds of goals are SECRET, PRECEDES, and AGREE.

The SECRET goal names a secret variable or attribute, like the group key, and requires that attacker be unable to obtain its value unless the attacker is (or has compromised) a group member playing one of the roles in the protocol. Current group members are defined by one or more parameters, one of which is a principal or group member, sufficient to identify the current group. For example, the GKA-S goal could be

```
SECRET Ks:  Last, mb_id,  or
SECRET Ks:  M.
```

In the first version, the value of K_s associated with the owner of a Last role may be shared with any other principal whose member state contains the same value of *mb_id*. In the second version, in any state with particular values of the attributes K_s and M (the whole member list), any element of M may share K_s. Both of these imply both forward and backward secrecy, since any user who is not (according to that user's own member state) a member of the present group view, identified either implicitly by *mb_id* or explicitly by M, is not permitted to share the current group key. For analysis, the policy would be reflected in the attacker model by allowing the attacker to hold any secret keys appearing as attributes of users not in the current group view. The analysis would include a sequence of group views showing members deleted and added.

The PRECEDES and AGREE goals state forms of authentication, based on agreement between parties on certain values. PRECEDES differs from

AGREE in that PRECEDES requires the other party to exist, and AGREE asks for agreement conditionally if the other party exists. Authentication goals would prevent a group member from using a key or token that was not sent by another member. They are sometimes necessary to exclude replay attacks, where an attacker may force an old key to be used repeatedly.

Extensions for new goal statements would be needed to express some substantively different ideas such as threshold secrecy (used in voting or fault tolerance), fairness, or anonymity.

5 Semantics and Analysis

Analysis of MuCAPSL specifications is made possible through translation of MuCAPSL to MuCIL (MuCAPSL Intermediate Language). MuCIL serves as an interface language through which several analysis tools can be made available.

5.1 MuCIL

In MuCIL (as well as in CIL) protocols are understood as specifying systems in which sending or receiving a message causes a state transition for some agent. Transitions are expressed as symbolic term-rewriting rules that can be used in different ways. They can be implemented in executable code, given adequate software library resources; or executed symbolically for security analysis by model checking; or viewed as logical statements for security analysis by inductive proof techniques.

Some security analysis approaches require "idealizing" the protocol actions into statements in a specialized logic. This is the case for BAN logic [4] and its successors. Where tools for automating BAN logic deductions are available, the idealization step is still manual. Furthermore, authentication proofs in this context do not establish the confidentiality properties that are central to multicast security, and this sort of analysis has not been done for multicast protocols. CAPSL permitted the user to supply the idealized logic version of protocol actions using ASSUME and PROVE statements. However, this facility was never applied in CAPSL, and it is not currently supported in MuCAPSL.

Transition rules are expressed in MuCIL as terms of the form

$$\texttt{rule}(\texttt{facts}(...), \texttt{ids}(...), \texttt{facts}(...), \texttt{if}(...)).$$

The "if" clause contains a boolean term and is optional. The left and right facts represent the prior and new state, respectively. The variables listed in the "ids" term represent nonces for which new, unused values are generated when the rule is executed. In general, facts are atomic formulas of the form $P(t_1, ..., t_r)$ where P is a predicate symbol and the arguments t_i are data terms. A data term is constructed from constants, variables, attributes, and function symbols that are defined in the protocol.

Facts express agent states or messages. There are two kinds of state facts: the "member state" of a group member, and the "protocol state" of the protocol process for a particular role. The current state of the system includes a member state fact and zero or more protocol state facts for each group member or potential group member, and also some message facts for messages that have been sent.

A protocol state fact consists of the group member agent object, a role, a state number and a list of terms that correspond to the agent's memory during the protocol execution. An example of the initial state fact of a group member M in the "Chosen" role is `state(M,Chosen,0,terms())` (the agent does not hold any protocol-specific terms).

A member state fact consists of the group member agent object and a list of terms representing group member memory components that persist across role instantions, including such data as the member's position and the size of the group. For instance, `member(M, terms(p,Np,Ks,M,KL,PT,FT))` is a member state fact for agents of type CLQ_CTX where M is a variable of type CLQ_CTX and p, Np, etc. are variables of the appropriate types as specified in the typespec CLQ_CTX in Section 4.1.

A message fact has the form `mmsg(terms)`. We do not keep sender or receiver addresses in message facts, or distinguish between unicast and multicast messages. This is motivated by the fact that these addresses make no difference from the viewpoint of security analysis, since an active attacker can always forge addresses or distribute unicast messages to the whole group.

When a rule is applied for a state transition, variables on the lefthand side of the rule are instantiated with values matching facts that currently exist in the system. All facts on the lefthand side are deleted from the current state configuration and replaced by the facts on the righthand side of the rule, instantiated with the corresponding values.

For illustrative purposes we present one rule to give a flavor of the semantics. The first event of the "Chosen" role, to receive new membership information, is interpreted by the following rule.

```
rule(
 facts(
  member(Mn,
   terms(p,Np,Ks,M,KL,PT,FT,mb_id
    vs_set,merge_set,leave_set)),
  state(Mn,Chosen,0,terms()),
  mmsg(terms(ID,M',VS,MS,LS)))
 ids(),
 facts(
  member(Mn,
   terms(p,Np,Ks,M',KL,PT,FT,ID
    VS,MS,LS)),
  state(Mn,Chosen,1,terms())),
if(Me=owner(Mn)).
```

A group member M_n in the initial state of the "Chosen" role can receive a multicast message with new values for the various membership attributes `mb_set, M, vs_set,...`. The condition tests that the name of the group member corresponds to the name stored in the attribute "Me." This is the assumption for that role.

5.2 Translation and Analysis

The primary task of the MuCAPSL-to-MuCIL translator is to translate the actions in the MESSAGES section of each ROLE specification into MuCIL rules. The translator achieves this by a two-step process: (1) parsing MuCAPSL messages into lists of events such as $\mathtt{send}(M, m)$ (unicast message), $\mathtt{mcast}(m)$ (multicast message), $\mathtt{recv}(m)$ (receiving a message), $\mathtt{new}(N)$ (generating a fresh value), or $\mathtt{eqn}(t_1, t_2)$ (test or assignment), and (2) translating each event into a rule, or sometimes a sequence of rules. Due to space limitations we cannot present the details of this translation here. The algorithm is described in [9].

The result of the translation is a MuCIL specification of the protocol that can then be used as input to any general purpose tool that analyzes state transition systems. To do security analysis, such tools generally require significant adapation that is independent of particular protocols but supports analysis strategies, cryptographic operations, and attacker models. To use MuCIL protocol specifications, an additional minor syntactic translator (a "connector") is needed to generate the corresponding tool input in its native language.

We have experimented with the Maude [13] tool for model checking the GDH.2 protocol suite. We added declarations of MuCAPSL data types and implemented general attacker rules in Maude that allow an intruder to duplicate, delete, or replay messages. The MuCIL specification of GDH.2 was entered into Maude by hand (because the connector was not ready at the time) and we used the Maude interpreter to execute it. For this purpose, we set up initial scenarios with a bounded number of members.

For the purpose of security analysis, we are interested in abnormal protocol behaviors. One way to detect such erroneous behavior is by defining states that violate some given policy. For instance, in the case of GDH.2 we expect that the result of any of the tasks is that the members of the group share a group key. Therefore, for one test, we defined an attack to be a protocol state in which at least two members of a group finished a specific protocol task but do not agree on the group key. Executing the initial configuration of the GDH.2 key distribution protocol (with well-behaving group members and an attacker) resulted in an attack state that is due to some confusion in the formats of messages in GDH.2. The attack takes less than a second to be found.

Searching for attacks is a bounded tree search that increases exponentially in time requirements with the size of the initial scenario, affected by factors such as the number of potential group members, the number of group views, and the number of roles. Most successful attacks seem to be demonstrable with small values of these parameters, but no guarantee of this can be made because the analysis problem is known to be inherently intractable in the worst case, even for unicast protocols.

To our knowledge there does not exist other comparable work for applying model checking tools to group multicast security protocols. We intend to perform further experiments and widen our approach to include theorem proving techniques to gain more confidence in our approach.

6 Concluding Remarks

We have attempted in this paper to motivate the design decisions that characterize MuCAPSL as a language for expressing group multicast protocols in a way that is suitable for application of formal methods, extends the CAPSL approach, is concise and still readable, has convenient syntactic features for group key distribution, and has a sound semantic basis in a term-rewriting model. We are still in the process of experimenting with the analysis of protocols expressed in MuCAPSL. While this work will primarily result in connectors and analysis techniques, it may also feed back into techniques for setting up security goals and environments in the most effective way to support such analysis.

References

[1] Y. Amir, Y. Kim, C. Nita-Rotaru, J. Schultz, J. Stanton, and G. Tsudik. Exploring robustness in group key agreement. In *Proc. of the 21st IEEE Intern. Conf. on Distributed Computing Systems, Phoenix, Arizona, April 16-19, 2001.*, pages 399–408, 2001.

[2] G. Ateniese, M. Steiner, and G. Tsudik. New multi-party authentication services and key agreement protocols. *IEEE Journal on Selected Areas in Communication*, 18(4):628–639, 2000.

[3] M. Baugher, T. Hardjono, H. Harney, and B. Weis. The Group Domain of Interpretation. Internet Draft, IETF, 2001. `http://www.ietf.org/internet-drafts/draft-ietf-msec-gdoi-01.txt`.

[4] M. Burrows, M. Abadi, and R. Needham. A logic of authentication. *ACM Transactions on Computer Systems*, 8(1):18–36, 1990.

[5] M. Clavel and J. Meseguer. Reflection and Strategies in Rewriting Logic. In J. Meseguer, editor, *Rewriting Logic and Its Applications, First International Workshop, Asilomar Conference Center, Pacific Grove, CA, September 3-6, 1996*, pages 125–147. Elsevier Science B.V., Electronic Notes in Theoretical Computer Science, Volume 4, `http://www.elsevier.nl/locate/entcs/volume4.html`, 1996.

[6] G. Denker. Design of a CIL Connector to Maude. In E. Clarke, N. Heintze, and H. Veith, editors, *2000 Workshop on Formal Methods and Computer Security (FMCS'00), July 20, 2000, Chicago, USA (post-CAV workshop)*, 2000. `http://www.csl.sri.com/~denker/pub_00.html`.

[7] G. Denker and J. Millen. CAPSL integrated protocol environment. In *DARPA Information Survivability Conference (DISCEX 2000)*, pages 207–221. IEEE Computer Society, 2000.

[8] G. Denker and J. Millen. Design and Implementation of Multicast CAPSL and Its Intermediate Language. CSL Report , Computer Science Laboratory, SRI International, Menlo Park, CA 94025, 2002. *To appear*.

[9] G. Denker and J. Millen. Modeling group communication protocols using multiset term rewriting. In *4th International Workshop on Rewriting Logic and its Applications (WRLA'02)*, 2002.

[10] B. Dutertre, H. Saïdi, and V. Stavridou. Intrusion-tolerant group management in enclaves. In *International Conference on Dependable Systems and Networks (DSN'01)*, pages 203–212, Göteborg, Sweden, July 2001.

[11] H. Harney, A. Colegrove, E. Harder, U. Meth, and R. Fleischer. Group Secure Association Key Management Protocol. Internet Draft, IETF, 2001. `http://www.ietf.org/internet-drafts/draft-ietf-msec-gsakmp-sec-00.txt`.

[12] Y. Kim, A. Perrig, and G. Tsudik. Simple and fault-tolerant key agreement for dynamic collaborative groups. In *Proc. 7th ACM Conference on Computer and Communications Security*, pages 235–244. ACM SIGSAC, 2000.

[13] Maude Web Site. `http://maude.csl.sri.com/`, 2000.

[14] C. Meadows. Experiences in the formal analysis of the GDOI protocol. Slides, Dagstuhl Seminar "Specification and Analysis of Secure Cryptographic Protocols, 2001. `http://www.informatik.uni-freiburg.de/~accorsi/dagstuhl`.

[15] J. Millen. A CAPSL connector to Athena. In H. Veith, N. Heintze, and E. Clarke, editors, *Workshop of Formal Methods and Computer Security*. CAV, 2000.

[16] J. Millen and G. Denker. CAPSL and MuCAPSL. *To appear in Special Issue of Journal of Telecommunications and Information Technology (JTIT)*, 2002.

[17] J. Millen and V. Shmatikov. Constraint solving for bounded-process cryptographic protocol analysis. In *8th ACM Conference on Computer and Communication Security*, pages 166–175. ACM SIGSAC, November 2001.

[18] S. Mittra. Iolus: A framework for scalable secure multicasting. In *Proceedings of ACM SIGCOMM '97*, September 1997.

[19] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In D. Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752. Springer-Verlag, 1992.

[20] O. Pereira and J. Quisquater. A security analysis of the cliques protocol suites. In *14th IEEE Computer Security Foundations Workshop*, pages 73–81. IEEE Computer Society, 2001.

[21] M. Steiner, G. Tsudik, and M. Waidner. Diffie-Hellman Key Distribution Extended to Groups. In *ACM Conference on Computer and Communications Security*. ACM, 1996.

[22] M. Steiner, G. Tsudik, and M. Waidner. Key agreement in dynamic peer groups. *IEEE Transaction on Parallel and Distributed Systems*, August 2000.

[23] C. Wong, M. Gouda, and S. Lam. Secure group communications using key graphs. *IEEE/ACM Transactions on Networking*, 8(1):16–30, February 2000.