Electronic Notes in Theoretical Computer Science 71 (2002)
URL: http://www.elsevier.nl/locate/entcs/volume71.html LastPage pages

Modeling Group Communication Protocols Using Multiset Term Rewriting [1]

Grit Denker [a,2,3] Jon Millen [a,2,4]

[a] *Computer Science Laboratory*
SRI International
Menlo Park, CA 94025, USA

Abstract

Protocols for secure group management are essential in applications concerned with confidential authenticated communication among coalition members, authenticated group decisions, or the secure administration of group membership 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) and its intermediate language MuCIL (MuCAPSL Intermediate Language). MuCIL is based on multiset term rewriting rules that permit state changes to be presented concisely, and in a way that closely matches the requirements of existing protocol analysis tools. With the help of the Group-Diffie-Hellman protocol suite we illustrate how secure group communication principles are modeled using multiset term rewriting.

1 Introduction

Reliable multicast protocols have been developed as a means to provide reliable ordered delivery of messages and membership services to a group of processes. One challenge in building a multicast protocol for use over a public network is security. Services such as confidential authenticated communication among coalition members, authenticated group decisions, or the secure administration of group membership and access control are at the core of secure and reliable group management. New protocols and frameworks have been designed to create multicast groups on a network and support secure group communication

[1] Supported by DARPA through SPAWAR Systems Center under Contract N66001-00-C-8014
[2] We thank Carolyn Talcott for helpful suggestions on an earlier version of the paper.
[3] Email: denker@csl.sri.com
[4] Email: millen@csl.sri.com

(e.g., GDOI [BHHW01], GSAKMP [HCH+01]). Some existing key exchange protocols for secure communication have been extended to the group setting (e.g., Group Diffie-Hellman GDH [STW96] and its authenticated form A-GDH [AST00]).

There have been only a few results on the formal analysis of group management protocols (e.g., Pereira and Quisquater analyzed A-GDH [PQ01] and Meadows discovered security flaws in early versions of GDOI [Mea01]). The analysis of group management protocols poses new challenges for formal analysis techniques. New language features and models are necessary to appropriately capture the concepts of such protocols. MuCAPSL (Multicast Common Authentication Protocol Specification Language) and its intermediate language MuCIL (MuCAPSL Intermediate Language) have been designed to meet these needs. The underlying design principles are

(i) Providing a high-level, yet mathematically well founded, protocol language that allows easy transformation of published descriptions of secure group communication protocols into the formal language

(ii) Providing a single common interface language that could be used as the input format for many formal analysis techniques or tools.

MuCAPSL and MuCIL are extensions of the CAPSL and CIL protocol analysis effort for unicast protocols [DM00a,DM00b,DM99]. MuCAPSL provides high-level specification concepts for multicast security protocols, such as message passing using unicast and multicast addressing, group membership data, and basic cryptographic operators. MuCIL is closer to state-transition representations of protocols. It serves two purposes: to help define the semantics of MuCAPSL in terms of multiset term rewriting, and to act as an interface through which protocols specified in MuCAPSL can be analyzed by using a variety of tools.

The emphasis in this paper is on MuCIL and its term rewriting semantics. We will illustrate the ideas of modeling group communication protocols with the help of the Group-Diffie-Hellman (GDH) protocol suite. An overview of GDH is given in Section 2. A semantic framework for multicast protocols is provided in Section 3. Rewrite rule generation is covered in Section 4. Section 5 describes future work.

2 Group-Diffie-Hellman Protocol

We illustrate the semantic model with the help of the Group Diffie-Hellman (GDH) protocol, which served as the basis for the Cliques protocol suite [STW96,STW98]. GDH [STW96] is an extension of the two-party Diffie-Hellman key agreement scheme to an arbitrary group size. The GDH protocol suite consists of a key distribution algorithm and protocols for member addition and deletion. For the purposes of this paper, we chose the key distribution protocol as an example because it incorporates unicast messages addressed to a particular group member as well as broadcast messages addressed to all group members.

The group key in GDH is computed from secret contributions from each group member. For this purpose, each group member M_i has a nonce N_i. Group member position numbers (the "i" in "M_i") are artifacts of the key distribution algorithm. They are somewhat stable but can change because of changes in group membership.

The group key is computed by raising the exponentiation base g to the product of all nonces $\Pi_{i=1..n} N_i$ of group members. The exponentiation base is known to every group member, whereas the individual nonces are secret to the particular group members. In a message exchange, members communicate partial key values that keep their secret and still allow other group members to compute a shared group key.

Figure 1 illustrates the communication between group members for a group of size 4. The member M_1 sends out an array consisting of the exponentiation base g and g^{N1} to its neighbor M_2. Every intermediate member receives such an array, multiplies each array element with its own nonce, and copies the first array element of the received message into its outgoing message. In this way, the length of arrays sent between group members always equals the position number of the receiving member. This "upflow" phase of GDH consists of unicast messages. Finally, the last group member receives an array of length n from which it computes the group key by raising the first array element to the power N_n. After this, M_n replies in a multicast to the group with an array of partial key values ("downflow" phase) that include its nonce N_n. The other group members can compute the group key from this multicast message by raising the appropriate array element to the power of their nonce. The intent of this protocol is that all group members share the group key $g^{\prod N_i}$.

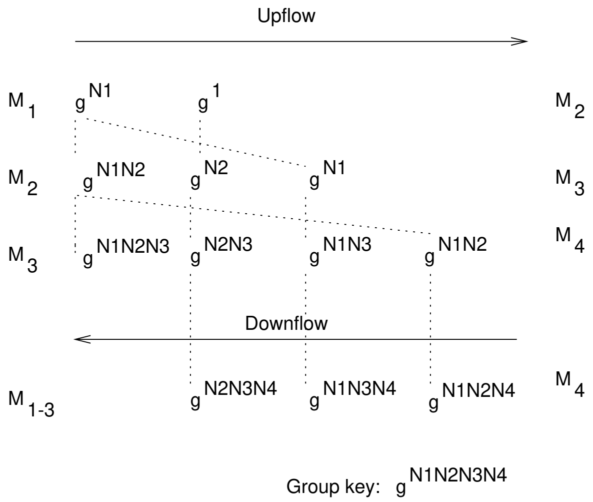

Fig. 1. Overview: Group Diffie-Hellman Key Distribution

2.1 Role Separation

In the GDH protocol, we can distinguish three different *roles*, in the sense of distinct behavioral sequences – the sequence of messages sent and received. The roles are M1, Mi, and Mn. M1 is the role of M_1, the group member who initiates the key distribution. The member M_n in role Mn is the last member of the group, the one whose position number equals the group size. All other members are in role Mi. (There is only one role Mi, not one for each i from 2 to $n-1$.) The roles are diagrammed schematically in Figure 2. A multicast message is indicated by terminating the message arrow with a box, as illustrated in the broadcast message from Mn. A message destination of "M" means that the role of its destination is ambiguous.

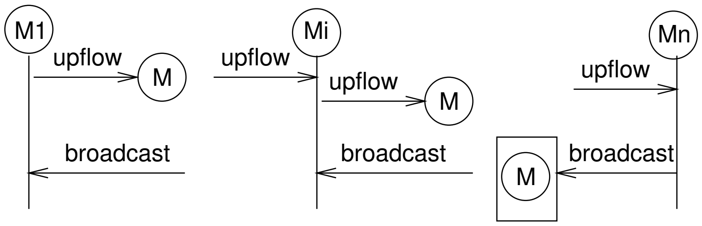

Fig. 2. Key Distribution Protocol – Overview

The schematic role diagram in Figure 2 can be elaborated by showing the contents of messages in more detail. This is done in Figures 3 and 4.

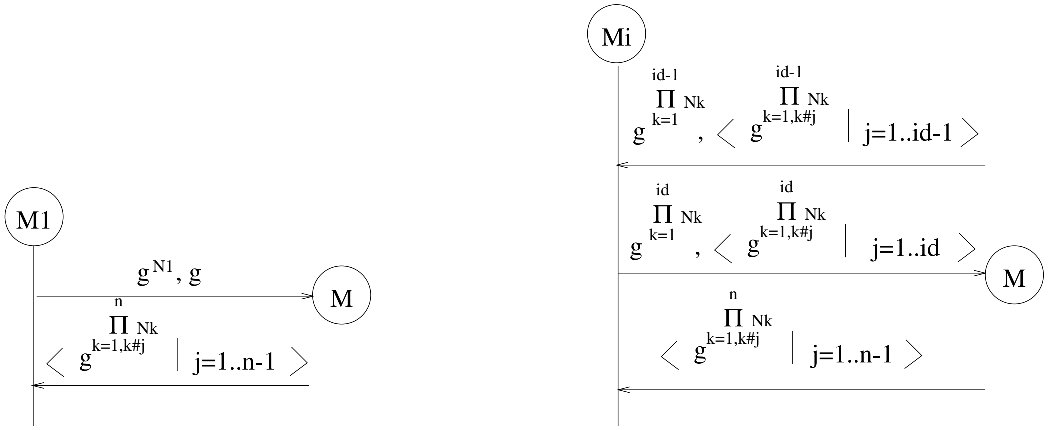

Fig. 3. Key Distribution Protocol – Roles M1 and Mi

These diagrams abstract away many important details, such as how received messages are stored, when new key contributions N_i are computed, and how messages are composed. Those details are part of the MuCAPSL specification and can be found in [DM02]. The graphical notation is only meant to give an overview of the relevant protocol roles and to illustrate the main message flow and message content.

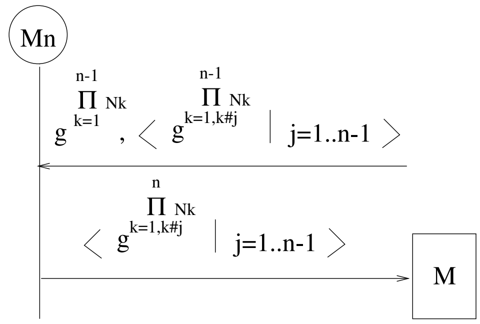

Fig. 4. Key Distribution Protocol – Role Mn

2.2 The MuCAPSL Specification

Since our focus is on term rewriting as in MuCIL, we do not include the entire MuCAPSL specification here. However, it may help to see a partial sample, so here is the role specification for M1.

```
PROTOCOL KeyDist;
  ROLE M1: GDHAgent;
    ASSUMPTIONS
      pos=1;
      HOLDS Nxt;
    MESSAGES
      NEW N;
      -> Nxt: g^N, g;
      <- KD;
      Kn=KD[1]^N;
  END M1;
  ...
END KeyDist;
```

In this specification, `KD` is the entire downflow array; only its first element is needed for M_1 to compute the new group key `Kn`.

3 Multiset Term Rewriting Semantics

As illustrated with the GDH protocol, there are a variety of different events in group communication protocols. These events include sending of multicast and unicast messages, receiving and checking the content of a message, and internal (cryptographic) computations that alter a group member's state. All these events must be modeled accordingly in the underlying semantics to capture the meaning of the group communication protocol.

In the MuCIL approach, protocols are understood as state transition systems. Each protocol event causes the system to undergo a state transition. A protocol is described through a set of transition rules of the form

$$F_1, ..., F_k \longrightarrow \exists X_1, ..., X_m : G_1, ..., G_n,$$

where each F_i and G_j is a "fact." Facts are atomic formulas of the form

$P(t_1, ..., t_r)$ where P is a predicate symbol and the arguments t_i are terms. A term is constructed from constants, variables, and function symbols. Variables, constants, and functions are typed. "Constant" and "variable" have their usual meaning for logical formulas; variables may be instantiated. Function symbols represent computations such as encryption used in the protocol.

In protocol modeling, facts are used to express the entrance of a process into a state, or the transmission of a message. The state of the network environment is a multiset of ground facts, and a rule is eligible to fire when the facts on the left side of the rule can be matched with facts in the multiset. When a rule fires, the matching facts in the multiset are removed from it and replaced by the facts on the right side of the rule, instantiated according to the substitution required by the pattern match. Removing a fact from the multiset reduces its multiplicity by one, if it was more than one.

Existentially quantified variables in the right-side facts are instantiated with new constants, as for a skolemization step in logic. In a cryptographic protocol model, these variables represent nonces, so it is essential to instantiate them with fresh values.

The origins of the multiset term-rewriting approach for cryptographic protocols appeared in [DMT98] and [CDL+99].

3.1 MuCIL Notation

As a language for expressing protocol transition rules, MuCIL has a particular syntax and a particular choice of predicate symbols used in facts.

Rules in MuCIL, as well as facts and terms, are expressed in functional notation representing a tree structure with labeled nodes having zero or more ordered children. There are two kinds of rules in MuCIL—unconditional rules of the form

`rule(facts(...),ids(...),facts(...))`

and conditional rewrite rules of the form

`rule(facts(...),ids(...),facts(...),if(...))`.

The "ids" term lists the quantified variables. The "if" clause contains a boolean term that may be complex, including conjunctions of boolean terms or other boolean operators.

Conditional rules were introduced for convenience. A single conditional rule can, in principle, be simulated by a sequence of unconditional rules, where the first transition creates a state that carries the result of evaluating the test condition, and the subsequent transition has a left-side fact that requires a true result. This sequence of two rules can be made atomic by a semaphore tactic. Conditional rules, however, are easier to produce and understand.

In this paper, for readability, we use the rule form *facts* $\longrightarrow$ ($\exists$ *vars*) *facts* `if` *test* in place of the official `rule(facts(`*facts*`),ids(`*vars*`),facts(`*facts*`),if(`*test*`))` (and similarly for the unconditional rule). MuCIL has a convention that only terms with plural function names (e.g., facts, ids, terms) have a variable number of arguments. The form `terms`(...) appears so often that, for readability in this paper, it is replaced by [...].

3.2 Facts

For multicast protocol modeling, three fact predicates are used: `mmsg` for messages, `state` for the state of a protocol role process, which we call an "agent," and `member` for the state of a group member.

A message fact has the form `mmsg`([...]), where the terms represent the message fields. We do not keep sender or receiver addresses in message facts because these addresses make no difference from the viewpoint of security analysis, since an active attacker can always forge addresses. There is also no difference between a unicast message and a multicast message, since an attacker can either rebroadcast a unicast message, or prevent any message, multicast or not, from reaching more than one, or any, destination.

Facts that represent the state of an agent in a protocol role have at least four arguments:

`state`(*ident, role, state-label,* [...]).

The first argument is the group member, the second is its role, the third is a sequence number or other label for the state, and the fourth is a list of terms representing the transient local memory of the agent, such as nonces, session keys, and the names of other group members.

A typical left-side agent state fact in a rule might look like

```
state(M1,KeyDistM1,2,[ ])
```

Here, `M1` is not a role name, but rather a variable of type GDHAgent, even though this identifier was borrowed from the role name given in the specification. The role name `KeyDistM1` is a constant produced automatically by combining the protocol and role names. We have to do this because role names are not unique across protocols. This example has no component terms because the role does not need any session-specific memory; it uses and modifies the group member memory only.

An important conceptual feature of state memory components is that their values are either undefined or "held," and, once held, cannot change. This is true because state variables in protocol specifications are thought of as having a single value for a protocol session, and then disappear with the termination of the protocol process when the session is ended. (In general, we use the term "state variable" to refer to the name used in a MuCAPSL specification for an agent state component.)

We cluster state component values into a single terms list in the fourth argument because those values are protocol specific, while every role of every protocol has the first three.

The state of a group member includes such components as the member's identity, position number, and the size of the group. The group member state fact has the form

`member`(*ident*,[...]).

The identity of a member is an object of a particular fixed type "GroupMember," which is defined in MuCIL to have `owner` and `groupid` attributes. Thus, the group member is an abstraction allowing a single user (of type Principal) acting as the owner to participate independently as a member of a number of different groups.

Because group member state components persist across protocol sessions, they are never considered undefined, but are always treated as held. However, unlike agent state variables, they may be updated during any protocol session. In fact, they could conceivably be updated several times by one role or concurrently by two or more agents of the same group member.

The state components after the member identity in the state depend on the group protocol architecture. The particular set of components for a given group is declared in a MuCAPSL specification as part of the definition of a subtype of GroupMember. Thus, we declare a subtype GDHAgent of GroupMember with the following list of named attributes: `pos, n, Kn, N, Nxt, KU, KD`. These store information, respectively, about the group member's position in the group, the group size, the latest group key, the secret nonce it contributes to the group key, the address of the upflow neighbor, and arrays for the upflow and downflow partial keys.

The names of the attributes are meaningful in the MuCAPSL specification, but not in the MuCIL rules. However, the translator attempts to make the rules it generates more readable by using attribute names (as well as state variable names) as the names of corresponding dummy variables in the rules. Thus, a typical left-side member fact in a MuCIL rule looks like

`member(M1,[pos,n,Kn,N,Nxt,KU,KD])`.

4 Events and Rule Generation

State transition rules are needed for protocol actions, attacker actions, and sometimes also for actions related to definition and detection of security goals. We only discuss protocol rules here, since the others are either not protocol specific, or they are constrained by the capabilities of different analysis tools.

A MuCAPSL group protocol specification usually specifies more than one protocol, since the same group responds to different protocols for such functions as key distribution, adding a member, removing a member, group split, or group merge. Within each protocol, there are several roles – three in the case of the GDH key distribution protocol, for example. Each role in each protocol is specified separately as a list of events involving messages and computational actions. For each role we generate a sequence of state transition rules, beginning with an initialization rule to create the first agent state.

In MuCAPSL we distinguish three different message events: receiving a message, sending a unicast message, or sending a multicast message. The concrete syntax of MuCAPSL events is not important here; it is covered in [MD02]. At an abstract syntax level, the three message events are

$$\texttt{recv}(\textit{terms(...)}),\ \texttt{send}(\textit{ident, terms(...)}),\ \texttt{mcast}(\textit{terms(...)})$$

As remarked earlier, MuCIL ignores the difference between a `send` and an `mcast`. The difference between a send and a receive is that a send generates a transition like

$$\texttt{member}(M, [...]), \texttt{state}(M, ...) \longrightarrow \texttt{member}(M, [...]), \texttt{state}(M, ...), \texttt{mmsg}([...])$$

and a receive generates a transition like

$$\texttt{member}(M, [...]), \texttt{state}(M, ...), \texttt{mmsg}([...]) \longrightarrow \texttt{member}(M, [...]), \texttt{state}(M, ...)$$

In both cases, the agent state label is incremented. A receive transition may also modify components of the member and agent states. Note that the member and agent state facts are tied together by reference to the same group member.

According to the semantics of rule firing, a receive rule removes the message fact from the multiset. If the message is supposed to be multicast, how does more than one group member receive it? There are three choices for the answer. One possibility is to change the rule to include the same message fact on the right, so that it will be preserved for other recipients. This redundancy was considered undesirable because it clutters the rule generator output. Alternatively, we could just make an exception in the MuCIL rule semantics for message facts. Finally, we observe that no exception is necessary to preserve state reachability, since the standard attacker is assumed to be able to duplicate messages.

MuCAPSL also has equational events *term* = *term*, which generate rules having no message at all, only a state modification. An equation may be either a test or an assignment. If it is a test, the equation reappears in the rule in the if-condition, and the only state modification is to increment the state label in the agent state. If it is an assignment, member and agent state components are modified.

A special event is needed to cause the value of an attribute to be updated with a fresh value, as required, for example, in GDH key distribution when a group member needs a new nonce. This is the event NEW X in MuCAPSL, or $\texttt{new}(X)$ in the abstract syntax. Fresh values are generated for state variables wherever they are first used, if they are not initially held or previously received.

MuCAPSL has a few limited constructs for flow of control. We tried to avoid supporting IF-THEN-ELSE, with the argument that when branching occurs, the protocol should end a role and start one of two new roles, with the choice information stored in an attribute. However, conditional branching, like other execution flow constructs, can be supported in MuCIL using rules that change the state label, with the existing fact types.

Something like DO-UNTIL is often needed in a multicast group protocol, although it does not show up in the GDH example. It is needed when the leader of a group must wait for responses from some designated number of other members before proceeding, to obtain a majority vote, or a consensus, or an agreement, or enough shares of a split secret to regenerate it.

An ABORT event is a way to bypass the normal requirement for a role—that it is expecting only one kind of message in each state. Some group protocols broadcast an abort message that halts whatever role is being played by each member. An abort message goes to a separate role, which issues an ABORT event to halt the existing agent or agents of that group member.

4.1 Implementability

The rule generator translates MuCAPSL events into rewrite rules. During this process, an implementability check must be passed that decides whether the specification is implementable.

The implementability algorithm determines what an agent must do to send or receive a message (multicast or unicast), and passes judgment on whether the explicit and implied actions are possible, given the current state of the agent. The state transition rules are, in effect, the product of the implementability algorithm. A failure of implementability implies that a rule cannot be generated that correctly models the event.

The key concepts for implementability are *computability, receivability,* and *invertibility.* A term is computable by an agent if the necessary variables are held (or can be generated on the fly, in the case of nonces or session keys) and all functions involved in complex terms are accessible by the acting agent.

Functions are accessible if they are public, which is the normal case, or private to the acting agent. Private functions are local to a user, but they are not state components because they are constant, and because they are often associated with the principal owning the group member rather than the group member. They are used to hold permanent information such as the long-term secret key of a principal, `sk`(A), which is private to A. Thus, `sk(Alice)` is computable by Alice but not by Bob. Private functions are declared with the PRIVATE property in MuCAPSL.

A term representing a message field is receivable if the agent holds enough variables and subterms to perform the implied pattern match. A pattern match on a complex term implies the ability to extract the components of it, by inverting the operation used to synthesize the term. Concatenations are always invertible, and encryptions are invertible if the proper key is computable. Thus, in order to receive a message specified as `ped(pk(A),X)`, the secret key `sk(A)` of A is needed, and the protocol would be incorrect if it specified that this message is received by a different principal B.

In the next sections we present some details on how each event is processed during rule generation. We start with initialization rules that are generated before processing any events.

4.2 State Initialization

State initialization rules are those rules that do not have a state predicate on the left side. There are initialization rules for member state facts and for agent state facts.

The translator generates initial agent state facts but not member state facts. A member initialization rule looks like

$\longrightarrow$ `member(M,[A,B,...])  if(...).`

where `M` is of a subtype of GroupMember having attributes `A,B,...`, and so on. We assume that there is only one member state per principal and group. This assumption is important because it eliminates ambiguity as to which member state is affected by role transition rules. This uniqueness assumption should be expressed by the if-condition. It is awkward to express because it refers to the entire multiset of facts. The rule generator does not attempt to produce member initialization rules.

There is one initialization rule per role to generate the corresponding initial agent state fact. In MuCIL we generate initial state facts with all state variables, some of which might be undefined in the initial state and will be assigned only in later steps of the protocol, as their values are generated or received in messages.

As it happens, no state variables are used in the key distribution protocol (defined or otherwise) and therefore the initial state predicate for the role M_1 is `state(M1,KeyDistM1,0,[])`.

A role specification may have "assumptions" that constrain initialization. There are two kinds of assumptions: "holds" assumptions that say that some state variables have meaningful initial values (but not specifying those values), and other assumptions, written as boolean expressions, that give or constrain values of attributes or held state variables.

An assumption on member attributes serves the purpose of identifying which roles it may execute. For example, in GDH, role M1 is the only role possible when $pos = 1$. Assumptions constraining attribute values become if-conditions for the initialization rule for that role.

Whether a state variable is initially undefined or held is important information for the rule generator, since an event that uses the value of a state variable is not implementable if that variable is not yet held. The rule generator knows which variables are initially held because they are assumed held, and it determines which initially undefined variables become held later as a result of protocol events. The held status of a state variable is not explicit in the rules, but it is implicit in the determination of which rules are implementable.

The initial rules for the three roles in the key distribution protocol, distinguished by the value of the position number attribute pos, are

```
member(M1,[pos,...])
⟶ state(M1,KeyDistM1,0,[ ]),member(M1,[pos,...]),
if(eqn(pos,1))

member(Mi,[pos,n,...])
⟶ state(Mi,KeyDistMi,0,[ ]),member(Mi,[pos,n,...]),
if(and(isgrtr(pos,1),isless(pos,n)))

member(Mn,[pos,n,...])
⟶ state(Mn,KeyDistMn,0,[ ]),member(Mn,[pos,n,...]),
if(eqn(pos,n))
```

The first and last could have been written with no if-condition by placing 1 or n, respectively, in place of pos, replacing the condition test with a pattern match.

One might also ask whether it is possible to create more than one agent instance for the same role. This is possible with the formalism but usually considered undesirable. The protocol specification can prevent concurrent role duplication by explicitly adding a semaphore attribute to the member state. This sort of concern is described further below when we discuss the ABORT operation.

4.3 Send and Mcast

It is easy to generate a rule that sends a message with terms $X_1, ..., X_n$. First, test that each of $X_1, ..., X_n$ is computable. If so, the rule simply increments the state label and adds the message to the right-side facts. The sending of g^N,g in GDH KeyDist role M1 produces the rule

```
member(M1,[..., N, ...])),state(M1,KeyDistM1,1,[ ])
⟶
member(M1,[..., N,...]),
state(M1,KeyDistM1,2,[ ]),mmsg([exp(g,N),g])
```

4.4 New

When an attribute receives a fresh value due to a NEW event or when a state variable that is a nonce is first used, the new value is created by a transition rule in which that attribute or variable appears in the existentially quantified list.

In the GDH example, the agent in role M1 must generate a new nonce N before it can send out the partial key to its neighbor. In MuCAPSL this is represented as NEW N, parsed to new(N).

The following is the rule generated for the NEW N event of the agent M1 in

the KeyDist protocol.
`member(M1,[..., N,...]))`, `state(M1,KeyDistM1,0,[ ])`
$\longrightarrow$ ($\exists$`N!`) `member(M1,[..., N!,...])`, `state(M1,KeyDistM1,1,[ ])`

Note that we had to invent a new dummy variable name $N!$ for the new value of N.

4.5 Data Types and Terms

Before we explain rule generation for other kinds of events, we need to say more about the construction of terms used in rule facts. Certain object types and function signatures are considered "built in" to MuCAPSL and therefore to MuCIL. The MuCAPSL translator is given a *prelude* containing abstract data type specifications for built-in symbols, and the user is allowed to supply additional declarations.

In the prelude, the most general object type is Object, and its most important subtypes are Boolean and Field. An object of type Field can appear as a message content field. Field has a subtype Atom containing familiar object types such as Principal, Nonce, and Nat (natural numbers), and some cryptography-specific types such as Pkey (public key). Nonatomic fields are formed, for example, by concatenation using the constructor `cat`. The prelude declares the public-key encryption operator `ped` and the symmetric-key encryption and decryption operators `se` and `sd`. Arithmetic operations are defined on type Skey for finite-length symmetric keys.

In multicast group protocols, there is often a need for variable-length messages or message fields, as well as a need for attributes holding an array-like sequence of values (e.g., arrays of partial keys in upflow and downflow messages of GDH). For this reason we have introduced the Array type and subtypes as required. Arrays are indexed by natural numbers, and their elements are of type Field. There are subtypes Sarray with elements of type Skey, and some others. (Array subtypes cannot presently be defined parametrically in MuCAPSL, so we need specific subtypes for arrays with different element types.) Arrays have operators such as *at* to extract the ith element, *with* to assign a value to the ith element, *proj* to project to a contiguous subsequence of an array, *acat* to concatenate two arrays, and *size* to determine the length of an array. There are elementwise operators to simplify common group computations, such as `A^^N` in MuCAPSL, with abstract syntax `aexp(A,N)`, creating an array of `A` values each raised to the power `N`.

It is also possible, in MuCAPSL, to define group member attributes that are functions, that is, they are indexed over an arbitrary datatype. For instance, a key table of a group member in which a shared secret key for every other group member is stored could be declared as a function

```
skTable(GroupMember):  Skey.
```

Including this function as an attribute allows us to reassign values for each table entry individually. Unlike arrays, functions are not objects of a datatype.

However, they are handled similarly for purposes of rule generation.

The next two sections explain rule generation for receive events and equational events.

4.6 Rule Generation for Receive

Receiving a message is conceptually complex in a language where the received terms are interpreted as patterns, and the state variables they mention may or may not be modified as a result of receiving the message.

Suppose that a protocol generates a nonce N which is sent first from user A, who generates it, to user B, and then returned from B to A. When B receives N, it is a new value that will be *stored* in B's state. When A receives N from B, the value received in the message will be *compared* with the value A originally generated, to test for possible errors or interference in the protocol. For state variables, one can determine which action is appropriate simply by observing whether the state variable is undefined or held. For attributes, either action is possible and might be intended in a given message, and the specification must indicate explicitly which action is intended. In MuCAPSL, the unadorned attribute N in a received message is supposed to be stored, while the syntactic form $?N$ indicates that the value of N in the message is supposed to equal the old value already stored, and a comparison test should be performed. The abstract syntax for the prefixed question mark is `compare( )`, but the "compare" is a translator directive that is not seen in the final MuCIL.

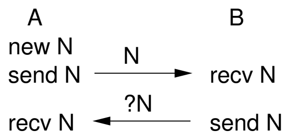

Fig. 5. Receiving

Thus, if N is an attribute, the event `recv([N])` is interpreted as an assignment. The following is a sketch of a rule implementing it, for some role `roleM` in state 4.

```
member(M,[...N...]),state(M,roleM,4,[...]),mmsg([N!])
⟶ member(M,[...N!...]),state(M,roleM,5,[...])
```

Note that we had to invent a new variable name `N!` for the value of `N` in the message.

If the receive event is `recv([compare(N)])`, the same rule is generated, except that there is no need for `N!`; it uses `N` everywhere. Thus, the state change consists only in incrementing the state label, and it occurs only when the comparison succeeds, that is, the message contains the value of `N` that is already in the member state.

If `N` is a state variable instead of an attribute, there are two differences: the `N` is found in the state fact instead of the member fact, and the rule generator can detect a mismatch between the action indicated by the use of "?" and the action implied by the undefined/held status of the variable. In the case of a mismatch, the rule generator can produce the correct rule and give the user a warning.

From the above discussion, it should be clear how to generate the rule for a receive of a single attribute or state variable, which may be qualified as needed by the "compare" operator "?".

Receiving a functional term

Receiving a function term `recv`$([f(s,t)])$ requires the agent to be able to pattern-match $f(s,t)$. This means that the receiver understands the structure of the term well enough to either duplicate the computation or learn the value of the atomic components it cannot compute. Algorithmically, this is expressed in the notions of receivability and invertibility.

One possibility is that the function value can be computed by the recipient from its own state and compared with the function value in the message. This is appropriate if all the variables appearing in the message term have the "compare" prefix. The result is a transition rule that changes nothing except the state label.

Otherwise, the function must be inverted. Invertibility of a function is expressed in MuCAPSL, as in CAPSL, by a special axiom. For example,

```
invertible(ped(pk(A),X),X,sk(A))
```

says that `ped` is invertible in its second argument `X` if the agent can compute the decryption key `sk(A)`. Concatenation is invertible in both arguments (with two axioms), and other functions (such as the hash function `sha`) are not modeled as being invertible and have no invertibility axioms. Thus, `sha(X)` is receivable only when the term is computable from the held value of `X`.

If a function term like $f(s,t)$ can be inverted by the acting agent to retrieve s or t, then receiving $f(s,t)$ reduces to receiving s or t or both, if possible. Thus, if `sk(A)` is computable, `ped(pk(A),X)` is receivable only if `X` is receivable, and the transition rule includes the effects of receiving `X`. A term like `cat(X,Y)` is invertible to both `X` and `Y`, and the transition reflects the recursively accumulated effects of receiving both, in left-to-right order.

Array and function attributes lead to special cases because of the peculiar property that one can modify a named complex object while providing only one component of it. The array reference expressed in MuCAPSL as `A[N]` with the abstract syntax `at(A,N)` is not invertible in the axiomatic sense, since neither the whole array `A`, nor (generally) the index `N`, can be determined from `at(A,N)`. Nor is the term computable, since it refers to a new value. (The old value would be referred to as `?A[N]`.) Yet this term should be considered receivable, since the receiving agent can store the value given in the message into the appropriate array element.

Receiving an event `recv([at(A,N)])`, if `A` is an attribute, results in a check whether N is computable, and generates a rule like

```
member(M,[...A....]), state(M,roleM,3,[...]), mmsg([V])
⟶ member(M,[...with(A,N,V)...]), state(M,roleM,4,[...])
```

where V in mmsg takes the place of at(A,N) in the message, and with updates an array by replacing one element at a given index. Notice that V is a new dummy variable of type Field. It is not allowable to receive an event at(A,N) if A is a state variable because state variables can be set only once, and that must be with an entire array object.

There are some other cases that we do not have the space to present here, such as the similar handling of function attributes.

4.7 Rule for Equational Event

An equational event $\mathtt{eqn}(x, y)$ can be a test or an assignment, depending on the left-side term x. The right side of the equation is always tested for computability. Generally, an equational event $\mathtt{eqn}(x, y)$ is a test, with the following exceptions in which it is interpreted as an assignment:

(i) $\mathtt{eqn}(X, t)$ where X is an undefined state variable

(ii) $\mathtt{eqn}(X, t)$ where X is an attribute

(iii) $\mathtt{eqn}(\mathtt{at}(A, i), t)$ where A is an attribute of type Array (or a subtype)

(iv) $\mathtt{eqn}(\mathtt{proj}(A, L, N), B)$ where A is an attribute of type Array and B is of type array

(v) $\mathtt{eqn}(F(x), y)$ where F is a function attribute

(vi) $\mathtt{eqn}(\mathtt{con}(x, y), z)$ or $\mathtt{eqn}(\mathtt{cat}(x, y), z)$, which get special treatment; see below.

Tests are translated into rules with the equation in the condition of the rule, similar to the rules for test in the previous section.

Rules for assignments of undefined state variables or atomic attributes are similar to the assignment rules for receive events in the previous section. The only difference is that there is no message fact on the left side of the rule since these assignments are internal state transitions.

An equational event with a concatenation on the left side produces two new events, namely, eqn(x,head(z)) and eqn(y,tail(z) in the case of con, and eqn(x,first(z)) and eqn(y,rest(z)) in the case of cat.

Events $\mathtt{eqn}(\mathtt{at}(A, i), t)$, $\mathtt{eqn}(\mathtt{proj}(A, L, N), B)$, and $\mathtt{eqn}(F(t), t')$ with nonatomic attributes A and F, respectively, are interpreted as assignments. The assignment of a term t to an array element $\mathtt{at}(A, i)$ is modeled by the following rule:

```
member(M,[...A...]),state(M,...,n,[...])
⟶ member(M,[...with(A,i,t)...]),state(M,...,n+1,[...]).
```

It is also checked that i is computable by the agent. The rule means that we substitute the old value at position i in attribute A with the new term t.

For the event $\texttt{eqn}(\texttt{proj}(A, L, N), B)$, we check that L and N are computable and replace the original array A with a new array that has been computed by projecting the original array A to two subarrays, the subarray from index 1 to index $L-1$, and the subarray from index $N+1$ to the end of A, and then concatenate those subarrays with B. The corresponding rule is
$\texttt{member}(M, [...A...]), \texttt{state}(M, ..., n, [...])$
$\longrightarrow \texttt{member}(M, [...A'...]), \texttt{state}(M, ..., n+1, [...])$,
where A' is defined as

```
acat(proj(A,1,pls(L,mns(1))),B,proj(A,pls(i,1),size(A))).
```

For an attribute F, the event $\texttt{eqn}(F(t), t')$ is handled similarly. Our solution introduces a predicate `update` for function symbols that is used to indicate that the value of a function is changed.
$\texttt{member}(M, [...F...]), \texttt{state}(M, ..., n, [...])$
$\longrightarrow \texttt{member}(M, [...\texttt{update}(F, t, t')...]), \texttt{state}(M, ..., n+1, [...])$.

However, this is cheating, because there is no function type in MuCAPSL. Function symbols like F do not have a type, so `update` is not a real function, because it does not have a legal signature. Depending on the expressiveness of a tool or analysis technique, means to handle `update` predicates in rules have to be found individually in the context of each analysis technique or tool.

4.8 Rule for Iterative Loop

The parser translates an iterative loop into $\texttt{dountil(events}(e_1, ..., e_k), c)$, where the sequence $e_1, \ldots, e_k$ of events is to be executed until the condition c becomes true.

The issue in a DO-UNTIL event is that one of the events e_i may set a previously undefined state variable during the first loop execution, but then the same event is a test on that variable in subsequent executions.

To handle this we create two state sequences for the loop when the state variable changes from undefined to defined in the body of the loop. The first sequence is used only once, and the second state sequence is used for all subsequent executions. This flow is illustrated in Figure 6.

The rule generator can tell whether a state variable becomes defined by saving the current state of all state variables at the beginning of the loop and comparing it with the current state after the first loop execution. Nested DO-UNTILs can be handled recursively.

4.9 Abort

The simplest form of role specification is a sequence of send and receive events. In these protocols, only one message type is expected in a given state. Sometimes it is desired to allow a group member or leader to multicast a special interrupt message telling other members to leave their current state and return to some stable state. Group members receive (or fail to receive) this message

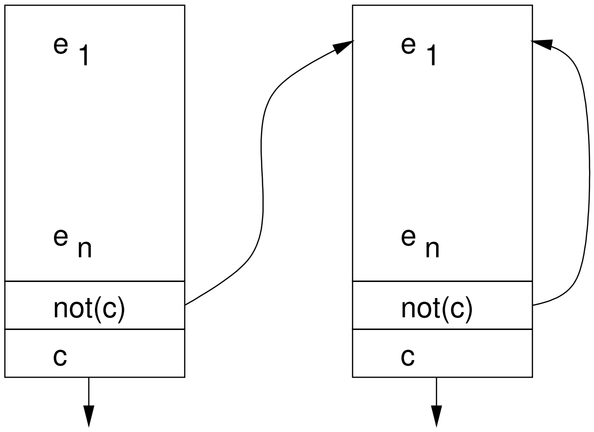

Fig. 6. DO-UNTIL Flow with Undefined State Variables

and act on it in some protocol-specific way.

This behavior is represented in MuCAPSL by adding an additional Abort role that begins by waiting to receive the interrupt message, which is an ordinary message of a protocol-specific format. The receive is followed by an ABORT event, which is then followed by any additional protocol-specific events executed under the assumption that the group member executing this role has terminated all of its other role agents.

The rewrite-rule version of this is simpler if a group member is single threaded, that is, it can execute only one agent at a time, other than the Abort role agent. The ABORT event in the Abort role is translated to a rule like

```
member(M,[...]),state(M,Abort,1,[ ]),state(M,R,N,T)
⟶ member(M,[...]),state(M,Abort,2,[ ])
```

where `M, R, N,` and `T` are all variables, so that any single nonabort agent state is matched and terminated. The single-threaded property can be guaranteed by adding an invisible semaphore attribute, and modifying initialization transitions (except for the Abort role) to test and set it, and terminating transitions to reset it. In MuCAPSL, giving a GroupMember subtype a MUTEX property causes the semaphore to be added by the translator.

It is also possible to implement an abort event when multiple roles, or multiple instances of any role, can be executed concurrently by a single group member. In that case the semaphore keeps count of the number of agents, and the abort role waits until the semaphore has reached zero before continuing.

5 Plans

To make MuCAPSL useful to protocol analysts, we need to make translation tools as well as documentation available. We are currently in the process of finishing the MuCAPSL-MuCIL translator that parses and typechecks MuCAPSL specifications and generates MuCIL rules for the protocol. In particular, we will add an optimization stage to combine successive MuCIL rules when possible, as we have done for CIL.

In the near future we will investigate goal specifications. Secrecy notions such as "perfect forward secrecy", and "forward/backward access control" need to be formalized in MuCAPSL and MuCIL. The capabilities of the translator for representing goals, execution scenarios, and an attacker model will be gradually expanded. Verification and model checking techniques have to be re-investigated and extended for group communication protocols.

References

[AST00] G. Ateniese, M. Steiner, and G. Tsudik. New multi-party authentication services and key agreement protocols. *IEEE Journal on Selected Areas in Communication*, 2000.

[BHHW01] 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`.

[CDL+99] I. Cervesato, N. Durgin, P. Lincoln, J. Mitchell, and A. Scedrov. A meta-notation for protocol analysis. In *12th IEEE Computer Security Foundations Workshop*, pages 55–69. IEEE Computer Society, 1999.

[DM99] 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/`.

[DM00a] G. Denker and J. Millen. CAPSL Integrated Protocol Environment. In D. Maughan, G. Koob, and S. Saydjari, editors, *Proc. DARPA Information Survivability Conference and Exposition, DISCEX2000, January 25-27, Hilton Head Island, SC, USA*, pages 207–222, 2000. `http://schafercorp-ballston.com/discex/`.

[DM00b] G. Denker and J. Millen. The CAPSL Integrated Protocol Environment. CSL Report SRI-CSL-2000-02, Computer Science Laboratory, SRI International, Menlo Park, CA 94025, October 2000. `http://www.csl.sri.com/~denker/pub_99.html`.

[DM02] 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.*

[DMT98] G. Denker, J. Meseguer, and C. Talcott. Protocol specification and analysis in Maude. In *Formal Methods and Security Protocols*, 1998. LICS '98 Workshop.

[HCH+01] 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`.

[MD02] J. Millen and G. Denker. CAPSL and MuCAPSL. *To appear in Special Issue of Journal of Telecommunications and Information Technology (JTIT)*, 2002.

[Mea01] 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`.

[PQ01] 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.

[STW96] M. Steiner, G. Tsudik, and M. Waidner. Diffie-Hellman Key Distribution Extended to Groups. In *ACM Conference on Computer and Communications Security*. ACM, 1996.

[STW98] M. Steiner, G. Tsudik, and M. Waidner. CLIQUES: A New Approach to Group Key Agreement. In *IEEE International Conference on Distributed Computing Systems (ICDCS'98)*, 1998.