SRI INTERNATIONAL

CSL Technical Report • February 1999

CAPSL and CIL Language Design

A Common Authentication Protocol Specification Language and Its Intermediate Language

Grit Denker and Jon Millen

Computer Science Laboratory

SRI-CSL-99-02, February 1999

Supported by DARPA through Air Force Research Laboratory under Contract F30602-98-C-0258

333 Ravenswood Avenue • Menlo Park, CA 94025-3493 • (650) 859-2000

Contents

Chapter 1

Introduction

1.1 Background

It is well known that protocols for exchanging cryptographic keys over data networks can be vulnerable to message modification attacks. This fact led to the development of tools for cryptographic protocol analysis. Some of the earlier papers on the subject are [MCF87, Mea91] on goal-directed state search tools implemented in Prolog, [Kem89] on the application of general-purpose specification and verification tools, [BAN90] on a specially designed logic of belief, and [Ros95, Low96] on the application of a model-checking tool for CSP specifications.

These tools and their successors have been effective, but it is difficult for analysts other than their developers to apply them. One reason for this difficulty is that the protocols must be respecified for each technique, and it is not easy to transform the published description of the protocol into the required formal system.

Some tool developers began work on translators or compilers that would perform the transformation automatically. The input to any such translator still requires a formally defined language, but it can be made similar to the message-oriented protocol descriptions that are typically published. The developers who took this approach were Millen, with an earlier version of CAPSL for the Interrogator tool developed at MITRE [Mil97]; Brackin, with

ISL, from which CAPSL initially borrowed much of its style, supporting an application of HOL to an extension of the GNY logic [Bra97]; Lowe, with Casper [Low98], for the application of FDR using a CSP model-checking approach; and Carlsen, who generated per-process CKT5 specifications from "Standard Notation" [Car94].

The idea of having a single common protocol specification language that could be used as the input format for any formal analysis technique was first presented at the 1996 Isaac Newton Institute Programme on Computer Security, Cryptology, and Coding Theory at Cambridge University. A version of CAPSL very close to the current one was subsequently implemented as an interface to the NRL Protocol Analyzer [BMM99].

The CAPSL language and supporting tools are still under development. This document offers a snapshot of the current design, not only for CAPSL itself, but also for the strategy by which CAPSL can be adapted for use by various protocol analysis tools. The core of this strategy is the use of an intermediate language, CIL, that is closer to the state-transition representation used by almost all of these tools.

1.2 CAPSL Specifications

The acronym "CAPSL" stands for "Common Authentication Protocol Specification Language." The language is intended to support analysis of cryptographic protocols using formal methods.

The core of a CAPSL specification is a message section showing the abstract format of a sequence of messages. Message fields are named and their types are indicated, but details such as field lengths and bit patterns are not shown. Only that information essential for protocol failure analysis is retained, resulting in a clear, simple model of the protocol.

Encryption operators, hash functions, and other computations are treated as abstract operators whose properties are specified axiomatically. Some standard operators, representing the abstract features of cryptosystems like DES, RSA, and Diffie-Hellman, are built in for convenience, and any others desired can be added, using auxiliary abstract data type specifications.

Between messages, one can place assignment statements and equations representing computations and tests. In this way, the computational notes usually relegated to informal text are formalized and incorporated into the specification.

An important part of the protocol specification is a statement of its security objectives. There is a "goals" section for this purpose, which may include secrecy and belief statements. Initial assumptions are also specified formally and placed in a section prior to the message list. It is possible to place assertions within the message list as well, to indicate intermediate goals or (to help support belief logic analysis) message idealizations.

Finally, there is also a way to specify scenario details to support tools that require setup of individual sessions such as finite-state search tools.

1.3 The Intermediate Language

The CAPSL Intermediate Language (CIL) serves two purposes: to help define the semantics of CAPSL, and to act as an interface through which protocols specified in CAPSL can be analyzed using a variety of tools.

The idea is illustrated in Figure 1.1. CAPSL is parsed and translated to CIL, and there are different translators from CIL to whatever form is required for each tool. CIL is designed to make the translation to tool-specific representations as easy as possible. The translator from CAPSL to CIL can deal with the universal aspects of input language processing, such as parsing, type checking, and unraveling a message-list protocol description into the underlying separate processes.

Fortunately, the protocol specifications required for most protocol analysis tools have considerable structural similarity. They generally specify a protocol with state-transition rules for communicating processes. An earlier suggestion for the CIL semantics [Mil95] used a small set of primitive operations (send, receive, assign, compare) to characterize state transitions. The new approach summarized in this document uses multiset term rewriting rules that permit state changes to be presented more concisely, and in a way that more nearly matches the requirements of analysis tools. This approach was influenced by an analysis example using Maude, by Denker, Meseguer,

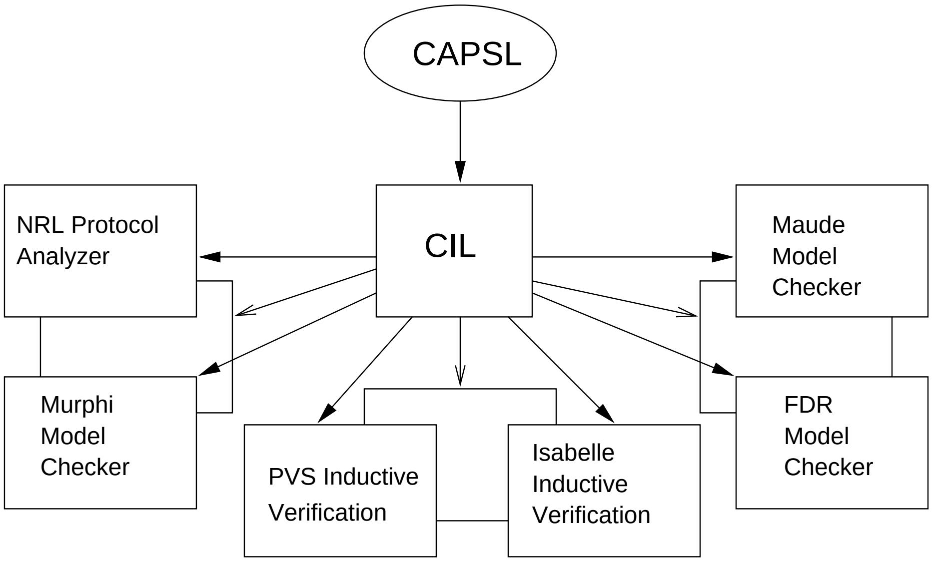

Figure 1.1: Overview of the integrated protocol environment

and Talcott, presented at a LICS '98 workshop [DMT98], and by Mitchell's linear logic formulation, presented at CAV '98 [Mit98], which is being used by Cervesato and Durgin at Stanford.

1.4 Summary of This Document

Chapter 2 presents the syntax of CAPSL, with a brief overview of the principal features of a CAPSL specification. CAPSL usage is illustrated with examples. A grammar for CAPSL is given in Appendix A.

The stages in translating CAPSL into CIL are summarized in Chapter 3. This includes the abstract syntax of CAPSL (given in full in Appendix B), the abstract syntax of CIL (given in full in Appendix C), and type checking, which produces a symbol table.

An abstract syntax is helpful as an intermediate stage because it should be relatively immune from the refinements that are expected to continue for the concrete syntax. It could also facilitate other approaches for protocol input, such as graphical interfaces.

Chapter 4 discusses the multiset rewrite rule approach to CIL. A formal algorithm for generating CIL from CAPSL is not given here; it will appear in a forthcoming document. Our purpose here is to outline the approach and illustrate it with examples. We will show examples of the transition rules that the translator should generate from the CAPSL declarations and message list. The translator must also deal with assumptions and security goals; in this document only a rough indication of the approach for this is given.

Chapter 5 has concluding remarks and summarizes the tasks that remain.

Chapter 2

CAPSL

2.1 Syntax Overview

A CAPSL specification is made up of three kinds of subspecifications: *type*, *protocol*, and *environment* specifications, usually in that order. Type specifications define cryptographic operators and other functions axiomatically, and are also used to define different types of principals. Environment specifications are optional; they are used to set up particular network scenarios for the benefit of search tools.

Here is a small example of a protocol specification, illustrating several of the language features. We will give a more extensive example later.

```
PROTOCOL One_Message;
IMPORTS USER;
VARIABLES
    A, B: User;
    K: SKey, FRESH, CRYPTO;
ASSUMPTIONS
    HOLDS A: B;
MESSAGES
    A -> B: {A, K}Pub(B);
GOALS
    SECRET K;
```

```
    BELIEVES B: HOLDS A: K;
END;
```

This protocol has only one message, in which user A sends a newly generated key K to another user B. The key K is concatenated with A and encrypted using B's public key.

Protocol variables are typed, and they may also have *properties*. For example, the property `FRESH` means that the value of the variable is a new value generated for each session, and has not (with cryptographically high probability) been used before by the principal generating it. The property `CRYPTO` means that the value of the variable is unguessable (because it comes from a large domain and is chosen in an unpredictable way). Properties affect interpretation of the message list and also affect the attacker model.

A specification may import another specification. Importing is understood as syntactically copying all definitions of the imported specification into the importing one. As a result, the defined types, functions, and axioms of both specifications are combined. In this example, the type specification `USER` (not shown) is imported by the protocol. Type specifications and environment specifications may also import other specifications. Assume for the example that the `User` type and the public-key function `Pub` are imported from the `USER` type specification.

The `HOLDS` declaration means that when an A session begins, it is initialized with the identity of a user B to communicate with. It is also implicitly initialized with the identity of its own user A.

There are two security goals: that the new key K is kept secret from any attacker (unless the attacker happens to be, or be colluding with, either of the named users A or B, or has subverted their software), and that if the B session completes, it shares K with A. (For this protocol, the goals would fail, since the message encrypted with B's public key could have been forged by an attacker.) More insight into exactly what these goals mean, and just what actions are performed by the session processes, can be found in Section 4.

2.2 Operators and Types

Messages in cryptographic authentication protocols are constructed using cryptographic operators and other functions, such as concatenation and hash functions. Every message field is of type `Field`, but certain operators require or produce fields of particular subtypes, such as key types.

In principle, all functions used in CAPSL and the data types they operate on must be specified axiomatically with abstract data type specifications (called *typespecs*). Several commonly used operators are defined in a standard prelude given in Appendix D.

The Prelude

Here is an example of two type specifications found in the prelude.

```
TYPESPEC PKEY;
TYPES Pkey;
FUNCTIONS
    keypair(Pkey, Pkey): Boolean;
AXIOMS
    keypair(J, K) = keypair(K, J);
END;

TYPESPEC SPKE;
IMPORTS PKEY;
FUNCTIONS
    ped(Pkey, Field): Field;
AXIOMS
    IF keypair(J, K) THEN ped(J, ped(K, X)) = X ENDIF;
    IF keypair(J, K) THEN ped(K, ped(J, X)) = X ENDIF;
    IF atomic(X) THEN atomic(ped(K, X)) ENDIF;
END;
```

The first typespec declares a data type Pkey (public key) and a symmetric relation `keypair`. The second typespec, which imports the first, builds on it by defining an encryption/decryption function `ped`. The axiom about `atomic` fields will be explained in Section 2.2.

Typespecs in the prelude are considered built in, and need not be supplied by the user or imported explicitly. The ones in the prelude state their imports explicitly just to make it clear what their dependency relationships are.

Data Types

The data types defined in the prelude are these:

Principal a persistent entity to whom messages are addressed.

Skey a key for any of the symmetric-key operators, `se`, `sd`, `sed`.

Pkey a key for any of the public-key operators, `pe`, `pd`, `ped`. Either the public or private key of a pair, indicated by the relation `keypair`.

Nonce a fixed-length field; assumed implicitly to have the `FRESH` property.

Field the universal supertype for message fields.

List the result of a list construction.

Timestamp a publicly held, ordered value that may be reused in other concurrent or very recent sessions.

There is also a Boolean type that is not intended to be used in messages; this is the value type for equations and other tests. All of these types except Field and List are subtypes of type Atom. Any field of type Atom has a well-defined length. This is important in decomposing concatenations, to be discussed in Section 2.2.

Typespecs must be supplied in order to use types or operators that are not included in the prelude, or if the user wants to override or extend the definitions given in the prelude. The prelude may be enlarged or modified in the future if that seems desirable.

One exception to the need for typespecs is that new primitive types used in a particular protocol may be introduced with a `TYPES` declaration in the protocol specification.

Operators

The operators defined in the prelude include

encryption: `se`, `sd` (symmetric-key encryption and decryption) and `sha` (hash) with key type Skey, and `xor` (bitwise exclusive-or) for combining Skey values; `ped` (public-key encryption/decryption) with key type Pkey

key agreement: `kap` (Skey to public value Pval) and `kas` (Pval and Skey to Skey) to create a common key

concatenation: `cat`, combining items of type Field, and `con`, creating values of type List

signature: `seal` (Pkey and Field to Pseal) to create a checksum and `verify` (Pseal, Pkey, and Field to Boolean) to check it

timestamp comparison: `later`, `near`, with Boolean values

CAPSL also allows finite-field arithmetic (+, -, *, /, ^, 0, 1, etc.) with type Skey and Boolean expressions with operators `AND, OR, NOT, IF...ENDIF, true, false`. These are not included in the prelude, but should be thought of as built in. Finite-field arithmetic (with an implicit, unspecified modulus) can be performed on values of type Skey.

Properties

Besides a type, a variable may have properties representing assumptions about the way it is generated or used. These properties generally do not affect the abstract model or representation of a protocol as a process, but they affect the feasibility of certain kinds of attacks.

The properties are

CRYPTO unguessable.

FRESH not reusable in other sessions by the same principal.

RANDOM unrecognizable, chosen with an approximately uniform distribution, to support conclusions about resistance to password guessing.

PRIVATE a function computable only by the principal given in its first argument.

EXPOSED a constant held by the attacker, or identifying a principal whose PRIVATE function values are computable by the attacker.

Functions declared `PRIVATE` are essentially local table lookups. Other functions declared in typespecs are public and can be computed by any party, including the attacker. This property is motivated by the agent model, explained in Section 2.5.

It has been suggested to offer "TRUSTED" or "HONEST" as a property of a principal, or "DISHONEST" for its negation. Because this is a property of individual principals rather than roles, CAPSL indicates dishonesty using an `EXPOSED` property on principal constants in the environment section, to indicate that their private information is compromised. See Section 2.6.

It seems convenient and efficient to list properties after the type, for example,

```
VARIABLES Ka:  Skey, CRYPTO, FRESH;
```

Concatenation

There are two kinds of concatenation in CAPSL: associative concatenation of fields, with the operator `cat`, and nonassociative concatenation to produce values of type List, using the operator `con`.

Concatenation with `cat` is supposed to represent simple bit-string concatenation. Its associativity leads to a problem in a language where pattern-matching is used to imply actions. For example, suppose X and Y are variables of type Field, and u, v, w are constants of type Field. Does $\texttt{cat}(u, v, w)$ match the pattern $\texttt{cat}(X, Y)$, and, if so, should we instantiate $X = u$ and $Y = \texttt{cat}(v, w)$, or $X = \texttt{cat}(u, v)$ and $Y = w$?

The older CAPSL document attempted to deal with this question by defining an equivalence relation "isom" between fields thought of as being of the same length. This approach did not support pattern-matching and led to complicated accessor functions. Here's a better approach.

There is a subtype Atom of Field, recognized by the predicate `atomic`. Most types – all types in the prelude except Field and List – are subtypes of Atom. Atomic fields have a well-defined length. This means that $\mathtt{cat}(X, Y)$ is decomposable uniquely into X and Y if and only if either X or Y is atomic. Thus, the pattern match to $\mathtt{cat}(u, v, w)$ results in $X = u$ when X and u belong to the same atomic type.

There are accessor functions `first` and `rest` on fields $\{u, v, ...\}$ that are well-defined when u is atomic; the value of `first` is u and the value of `rest` is $\{v, ...\}$.

Concatenation with `con` constructs values of type List, which is like a LISP dotted pair. It is not associative. It represents a kind of concatenation where component delimiters are explicit and nestable in the concatenated field.

There are accessors `head` and `tail` for `con`, such that `head(con(A, B)) = A` and `tail(con(A, B)) = B`. A non binary list like `con(A, B, ...)` is, by convention, interpreted right-associatively as `con(A, con(B, ...))`.

Bracket Notation

The expression `{A, K}Pub(B)` is interpreted as `ped(Pub(B), cat(A, K))`. This is actually a combination of two notational conventions, namely the use of `{A, K}` to abbreviate `cat(A, K)` and the postfix of a Pkey term to a concatenation to encrypt it using `ped`. Similarly, `{X, Y}K` is interpreted as `se(K, cat(X, Y))` if K is of type Skey. The brackets also indicate encryption even without a concatenation, so that `{X}K` is interpreted as `se(K, X)`. Decryption with `sed` is indicated with a prime, for example, `{X}'K`.

The other concatenation operator, `con`, is indicated using square brackets. Thus, `[A, B]` is interpreted as `con(A, B)`, and encryption may be applied with postfix keys as for `cat`.

Perhaps CAPSL should have a meta-construct for reassigning the postfix notation to other encryption operators.

2.3 Messages

The `MESSAGES` section contains messages and actions. A message is expressed in the form

$$source \to destination:\ field_1,\ \ldots,\ field_n$$

where the source and destination (sender and receiver) are variables of type Principal. Messages may be numbered.

Messages may be preceded and followed by equational actions representing assignments or tests. A *phrase* consists of a message and the actions associated with it. An action prior to a message is performed by the sender of the message, and an action after the message is performed by the receiver.

If the receiver of a message is the sender of the next message, any action between the two messages is performed by that principal, and it does not matter which phrase the message belongs to. If not, the association of actions with messages can be indicated with /, as in this example:

```
A -> B: X;
X = Y;/
A -> C: Z;
```

Here the / shows that the action `X = Y` is performed by B rather than A, and it belongs in the phrase containing the first message.

Actions

An equational action like `X = Y` may be either an assignment to X or a comparison test; it depends on whether or not the acting principal already holds a value for X. As an action, a comparison test is evaluated at runtime (the CAPSL translator does not attempt to determine whether it must succeed or fail). It has no effect if it succeeds. If it fails, the effect is to halt the acting principal, that is, terminate the process executing its role in the protocol.

The right side of the equation may be any term that is computable by the acting principal (because it holds the term or all its components).

The left side of the equation is normally a variable, but it might also be any computable term, if a comparison test is intended. It can also be a concatenation, like {`X, Y`} `= Z`, in which case it is an abbreviation for the two actions `X = first(Z)` and `Y = rest(Z)`.

Assignment actions do not have to be placed in the message list if they have been placed in a `DENOTES` declaration. The CAPSL translator will look for them there if it cannot otherwise determine that a certain variable or term is held by the sender of a message containing it. If more than one equation for the same variable is there, the translator should look at all of them to find one that works for the agent that needs it. A `DENOTES` declaration is a convenient way to prominently exhibit the format of conceptually important items such as tickets and certificates.

A Boolean-valued term may also be used as an action, if it is computable by the acting principal, since it defines a test.

Use of %-Notation

Sometimes it is necessary to distinguish between the sender's and the receiver's view of a message, because the receiver may have more or less knowledge about the structure of a message field than the sender. For this purpose, we exploit the %-notation introduced in Casper [Low98].

A %-term is a term of the form $u\%v$, or a term containing some subterm of that form. In the %-term $u\%v$, u is the sender's version of the term and v is the receiver's version. A term like {`A%B, C%D`} makes sense, since the sender constructs {`A, C`} and the receiver sees {`B, D`}, while a term like `A%`{`C%D`}`K` does not make sense. No % should be within the scope of another. Any %-term could be written equivalently with only a single top-level use of %. The precedence of % is lower than that of any other operator except the comma that separates message fields.

Assignments can be used to avoid %-notation, if desired. For example, the message

```
1.  A -> B: A, {kap(Ra)}K%{PA}K;
```

could be expressed in three steps as

```
    M = {kap(Ra)}K;
1.  A -> B: A, M;
    PA = {M}'K;
```

The CIL output of the three-step version will have more steps and less efficient use of pattern-matching. This makes it less suitable for most kinds of analysis, but more suitable for generating prototype code.

2.4 Assertions

CAPSL permits a variety of assertions to be stated about operators and protocol variables. Axioms in a typespec are universal assertions about the operators defined there. Assumptions and goals in a protocol are statements about protocol variables; they refer implicitly to the current state of the network, including the local states of the communicating agents. (An *agent* is a process running on behalf of a principal in some role; see Section 2.5.)

Assumptions and Goals

The `ASSUMPTIONS` section in a protocol states initial conditions. An important use for this section is to indicate which variables are held (their values are known) initially by each principal. This is done with statements like `HOLDS A: B, S`.

The `GOALS` section states the security objectives for the protocol. A goal referring to the state of an agent should be true for the final state of that agent.

Assertions may also be given as axioms in a type specification or in an environment specification.

Assertion Keywords

There are keywords to express `SECRET`, `BELIEVES`, `KNOWS`, and `HOLDS` assertions.

HOLDS for a principal: the variables whose values it holds, for example, `HOLDS A: Ka`.

SECRET for a variable: the principals who may hold its value, for example, `SECRET Ka: A, S`. Used without a list of principals, it applies to all principals named in the protocol.

SESSION_SECRET for a variable: the principals who may hold its value during a session. For example, `SESSION_SECRET R: A, B`. Session secrets are not sensitive after the session has terminated.

BELIEVES for a principal: its initial beliefs. Any statement may be believed by any principal. For example, `BELIEVES A: HOLDS B: K`.

KNOWS same syntax as for believes. A principal knows an assertion if both (1) the principal believes the assertion and (2) the assertion is true.

Principals are implicitly assumed to hold their own identities, so `HOLDS A: A` is unnecessary. Freshly generated variables (nonces) may either be assumed held initially or not, as a matter of style, or the choice may be forced by the capability of the analysis tool used. Timestamps are implicitly assumed held by all principals.

Many authentication goals can be expressed using `HOLDS` statements, since `HOLDS B: X` implies agreement between B's value of X in B's final state and the original value of X initially held or generated by whichever agent introduced it.

If a variable is asserted `SECRET` in the `GOALS` section, it is unnecessary to assert it `SECRET` in the `ASSUMPTIONS` section as well; that is implied. The same goes for `SESSION_SECRET`.

If no principals are named in a `SECRET` statement, it means that the secret item should not be obtainable by the attacker in any session where none of the roles is overtly played by the attacker.

Besides the assertions that use the keywords, an assertion can be any statement with a logical or Boolean value, such as an equation or predicate or Boolean function call.

Intermediate Assertions

Statements intended as assumptions or intermediate proof obligations may be placed among the actions preceding or following a message. Assumptions should be preceded with the keyword `ASSUME` and proof obligations should be preceded with the keyword `PROVE`, to distinguish these statements from each other and from executable tests.

Either type of statement is referred to as an *intermediate assertion.* An assumption preceding a message might be intended as the belief-logic idealization of the message. However, idealizations are sometimes also handled as proof obligations, so probably some additional syntax for them is called for.

2.5 The Agent Model

Protocol specifications in CAPSL, like most textbook protocol specifications, are written from the point of view of a single protocol session. Two or three principals are conversing among themselves according to the given message sequence and formats. Each protocol variable of type `Principal` has a role implicitly associated with it. Roles are specified in a context that is oblivious to attacker interference, or the existence of other, concurrent sessions of the same or different protocols.

Agents and Principals

Models for analyzing attacks generally require a more explicit awareness and representation of the network environment, which includes the attacker and multiple sessions. Other sessions may have been completed in the past, or they may be running concurrently. Each participant in the protocol is a process, called a *session process* or *agent,* which is characterized both by its role in the protocol and by the principal on whose behalf or authority it operates. Agents are transient entities that terminate at the end of a session (or sometimes earlier).

A principal (that is, an object or instance of type Principal) is a persistent entity. Principals may be users (Alice, Bob) or long-lived well-known processes (e.g., a database server or key distribution center). The capability and authority of a principal rests in the set of keys it holds, especially the long-term (as opposed to short-term or per-session) keys.

One important point about principal instances as persistent entities is that they may have many agents acting on their behalf, in the present or in the past, playing the same or different roles in the protocol. Given a protocol specification with protocol variables A and B of type Principal, one can for instance instantiate A with Alice and B with Bob in one session, and in another session A is instantiated with Bob and B with Alice.

All agents of the same principal share the same long-term keys and other such information. This is relevant for analysis because it means that messages or message fields from one protocol session may be using the same long-term keys as those in another session. Such messages are potentially useful in an attack that involves replaying information from one session into a different session.

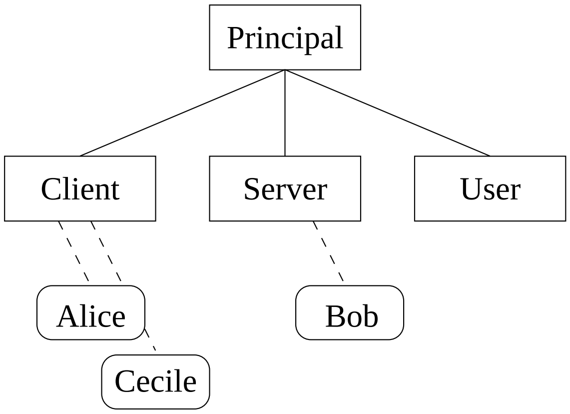

Figure 2.1: Agent Model

The type Principal is primitive and has no functions defined as key attributes. However, different subtypes of Principal may be defined. Principal subtypes are distinguished by what long-term information they possess. The subtypes needed depend on the protocol. A protocol using a symmetric-key system may, for example, divide principals into clients, each of which has one key shared with a server, and servers, which have an array of keys, one for each client. Figure 2.1. shows how the Principal type may have subtypes defined for it, and these have instances such as Alice and Bob. Client, Server, and

User are principal subtypes, Alice and Cecile are clients (constants of type Client), and Bob is a server.

Typespecs for Principal Subtypes

Type specifications are the means to define subtypes of Principal together with their long-term attributes (see Section 2.8). A typespec may define one or more types; here is an example defining Client and Server.

```
TYPESPEC Agent;
TYPES Client, Server: Principal;
FUNCTIONS
  PW(Client): SKey, PRIVATE, CRYPTO;
  CP(Server, Client): SKey, PRIVATE, CRYPTO;
AXIOMS
  PW(C) = CP(S, C);
END;
```

A client has only its own password `PW(C)`, while each server possesses a table `CP(S, C)` of passwords (one for each client). Both these functions are declared to be `PRIVATE` and `CRYPTO`. Private functions have a first argument identifying a principal that can access that "row" of the function. Thus, since the password function `PW` is private, the value `PW(C)` is available only to the client C. In particular, the attacker can only see `PW(X)`, where X is some agent identity owned or compromised by the attacker. Furthermore, it would be an error for a protocol specification to indicate that `PW(C)` is used in an action or transmitted message by a principal in a different role, say B.

Constants of principal subtypes (principal instances) are specified in environment specifications. The mapping between principal instances and the roles of a protocol, for each session, is defined in the `SESSION` sections of the environment specification.

2.6 Environment Specification

When a protocol is being analyzed or simulated, the analyst may have to specify which agents are to be run. The analyst may also have to supply other run-specific information such as the initial knowledge of the attacker. It was envisioned that CAPSL specifications could include environment specifications containing this kind of information. Environment specifications are outside the protocol specification, like typespecs.

The content and interpretation of an environment section depends on the analysis tool. However, it seems desirable for CAPSL to provide syntax, keywords, and organization so that different tools can at least take advantage of the CAPSL parser.

Declarations to name principals and other constants could be placed in this section, and sessions can be defined. For example, suppose we want a session in which Alice takes the client role and Bob takes the server role. We might set that up like this (assuming a protocol specification `Protocol` with variables `A: Client, B: Server`):

```
ENVIRONMENT Test1;
IMPORTS Protocol;
CONSTANTS
    Alice: Client;
    Bob: Server;
SESSION S12;
    A = Alice;
    B = Bob;
EXPOSED
    CP(Bob,\_);
END;
```

Sessions are specified here by associating roles with agent constants. The role names are the variables of any subtype of `Principal` who occur in the imported protocol specification. Any other variables that are initially held by any principal must also be assigned here. It is because the protocol variables are used here that the protocol specification is imported.

The initial knowledge of the attacker is in the `EXPOSED` section. This would normally be a list of terms that the attacker is assumed to hold initially, possibly including some items that are declared in the protocol as secret. The attacker may be implicitly assumed to hold agent names (if they are not defined to be `CRYPTO`).

We could also allow an exposed list item like `CP(Bob, _)`, with an underline for an anonymous variable, to mean that a whole row of one particular function is assumed compromised.

If a constant of type `Principal` is declared as `EXPOSED`, then its row of any private function is exposed. Thus, if `Bob` is exposed, then `CP(Bob, _)` is automatically exposed. A "dishonest" principal (in the NRL Protocol Analyzer sense) could be declared this way.

The `SESSION` statement implicitly defines `S12` as a constant of the session identifier type SessID. This type is a built-in subtype of Nonce.

There may be more than one session defined. Sessions are, by default, assumed to run concurrently. It is also possible to express execution dependencies between sessions. For this purpose we provide an `ORDER` section to specify a series-parallel sequencing of sessions. Some search tools could benefit from this and save time when such a restriction is assumed. For example, we might say `ORDER (S11;S2)||S13` to mean that session `S12` does not start until `S11` ends, but that sequence runs concurrently with `S13`.

2.7 Subprotocol Invocation

In larger protocols it is sometimes convenient to recognize functionally related groups of messages called "exchanges" or subprotocols. It makes sense to identify subprotocols in the CAPSL specification as well.

Somewhat different concepts are involved, depending on whether a subprotocol is in fact designed to be subordinate to a larger protocol, or whether one is combining two independent protocols with a simple data flow relationship in the context of a larger system. The latter case is not presently handled well by CAPSL. Some suggestions are given here for subordinate subprotocols.

Subprotocols are usually offered as options to be chosen or negotiated, or applied in sequence. In CAPSL, one can replace a message with a conditional selection or a subprotocol invocation.

There are some complications in understanding the meaning of these language features, which will be addressed as we determine how to generate the CIL rules for them. Some of the difficulties are (1) how to resolve conflicts in variable name declarations in different protocols, (2) how to synchronize the case splits in different agents after a conditional selection, and (3) how to merge different branches of a protocol into a common continuation.

The intent of the current grammar is that subprotocols be declared in separate `PROTOCOL` specifications, which are invoked by name in the parent specification by using `INCLUDE` statements. The child protocols import the parent protocol (not vice versa!), so that they can use the protocol variables. The parent protocol may refer only to the names of the subprotocols. Analysis is simpler if we assume that subprotocols do not "return," they just take over the parent session.

Subprotocol invocation is illustrated in Appendix E for the SSL handshake protocol.

2.8 EKE Example

Here, we illustrate the different components of a CAPSL specification in more detail with the help of an example. This is the basic "encrypted key exchange" (EKE) protocol, due to Bellovin and Merritt [BM93]. We will refer back to this example in later sections to discuss its translation into CIL.

In the EKE protocol a client and a server exchange messages with the aim of both holding a shared, secret key at the end of the protocol. The secret key is computed by using freshly generated nonces and keys, symmetric-key encryption, and Diffie-Hellman key agreement.

Type Specification

This version of the EKE protocol uses clients and servers as principal subtypes. The difference is that a client has only its own password, while each server possesses a table of passwords (one for each client). Both these functions are declared to be PRIVATE and CRYPTO.

```
TYPESPEC Agent;
TYPES Client, Server: Principal;
FUNCTIONS
  PW(Client): SKey, PRIVATE, CRYPTO;
  CP(Server, Client): SKey, PRIVATE, CRYPTO;
AXIOMS
  PW(C) = CP(S, C);
END;
```

Note that variables in typespecs do not have to be declared. If their declarations are missing, the CAPSL compiler will automatically infer their types from the signatures of the functions for which they are used as arguments (and generate an error message if they are used inconsistently).

Protocol Specification

The EKE protocol imports the Agent type specification. It also uses key agreement operators, an abstraction of the Diffie-Hellman system, which are implicitly imported from the KeyAgreement typespec in the prelude. The operation `kap` takes an Skey and generates a public value, which can be combined using `kas` with another Skey to yield a new shared Skey.

```
PROTOCOL EKE;
IMPORTS Agent;
VARIABLES
    A: Client;
    B: Server;
    Ra, Rb: SKey, FRESH, CRYPTO;
    Na, Nb: Nonce, CRYPTO;
```

```
    PA, PB: Pval;
    K: SKey;
ASSUMPTIONS
    HOLDS A: B;
MESSAGES
1.  A -> B: A, {kap(Ra)}PW(A)%{PA}CP(B, A);
     K = kas(PA, Rb);
2.  B -> A: B, {kap(Rb)}CP(B, A)%{PB}PW(A),{Nb}(K%kas(PB, Ra));
    K = kas(PB, Ra);
3.  A -> B: A, {Na, Nb}K;
4.  B -> A: B, {Na}K;
GOALS
  SECRET K;
  BELIEVES A: HOLDS B: K;
  BELIEVES B: HOLDS A: K;
END;
```

CAPSL assumes that each principal holds its own name, by default. So, for example, the assumption `HOLDS B: B` is unnecessary.

The use of the % notation in this protocol is of interest. Here, A sends B two items: its own address and the term `{kap(Ra)}PW(A)`. What is the receiver's view of this term?

Since `PW` is a private function, B cannot access `PW(A)`. Instead, B will try to decrypt the message by using a key that is accessible to B. `CP(B, A)` is private to B and `CP(B, A) = PW(A)` due to the agent type specification. If the use of `CP(B, A)` were not indicated, the CAPSL translator would have to look among the typespec and protocol assumptions to find one (namely, `CP(B, A) = PW(A)`) proving that B holds an appropriate key. The initial version of the CAPSL translator might not be smart enough to do that, so it is helpful, if not theoretically necessary, for the specifier to show the receiver's version of the key.

We note also that B cannot determine whether the value inside the encryption is of the form `kap(Ra)`; it can only view it atomically in the form `PA`. If the receiver's version were left as `kap(Ra)`, the CAPSL translator should generate an error message because the receiver cannot check the pattern.

Environment Specification

There is no fixed environment specification for a protocol; it is designed to meet the needs of the analyst. To illustrate the syntax, suppose we want a session for EKE in which Alice takes the client role and Bob takes the server role. We might set that up like this:

```
ENVIRONMENT Test1;
IMPORTS EKE;
CONSTANTS
    Alice: Client;
    Bob: Server;
    Pa: Skey;
AXIOMS
    Pa = PW(Alice);
SESSION S12;
    A = Alice;
    B = Bob;
EXPOSED Pa;
END;
```

A constant `Pa` assumed equal to `PW(Alice)` is declared as `EXPOSED` just to show the associated syntax.

Chapter 3

Translator Stages

3.1 Translator Overview

The translator from CAPSL to CIL has some commonplace tasks to perform, like parsing and typechecking, and it also performs the conceptually challenging task of creating a state-transition model of the protocol based on a pattern-matching rewrite-rule approach. Chapter 4 is dedicated to the topic of rewrite rule production. This chapter presents an overview of the translation process and describes the abstract syntax of CAPSL and CIL.

The translation has the following steps:

1. parsing to CAPSL abstract syntax (see Section 3.2)
2. typechecking to produce a symbol table (see Section 3.4)
3. minor transformations (see Section 3.5)
4. state transition rule and slot table generation (see Sections 4.1 - 4.6)
5. assertion processing (see Section 4.7)
6. attacker rule generation (see Section 4.10)

The CAPSL abstract syntax represents an intermediate processing step (see Section 3.2).

Translation steps 2-5 contribute to the CIL representation of a CAPSL protocol. In Section 3.3 we give an overview of the CIL abstract syntax.

A generalized attacker can also be incorporated into the multiset model, and it could be customized using information in the environment section. An idea of how that could be done is described in Chapter 4.

3.2 CAPSL Abstract Syntax

The abstract syntax form of a CAPSL specification represents an intermediate phase of syntactic processing. In this form, the specification has been reduced to its parse tree. The parse tree is represented linearly using the abstract syntax, which is simpler than the concrete syntax. Keywords and punctuation have been eliminated, and there are fewer nonterminal symbols. The abstract syntax is useful as a starting point for type checking and other semantic checks and translation, and it remains stable over inessential language modifications such as changes in keywords.

A context-free grammar for the CAPSL abstract syntax is given in Appendix B. In this section we give the abstract syntax representation of the EKE example. The representation is a linear (with line breaks and indentation for readability) encoding of the parse tree.

Each subtree of the parse tree is in one of four forms:

- *node-name(subtree, ...),*
- *[subtree, ...],*
- *functional expression,*
- *identifier.*

A node name corresponds to (but is not necessarily identical to) a nonterminal symbol in the CAPSL grammar, and has a fixed number of subtree arguments. A bracketed list represents a node below which there is a sequence of subtrees that may be empty or arbitrarily long.

The abstract syntax representation below is for the `Agent` typespec followed by the EKE protocol and then the environment specification. Note that there are `crypt` operators arising from encryption that have yet to be resolved into `se` or `ped`, since that requires a type check.

```
specification(

typespec(Agent,
  decls([
    types([Client,Server],Principal),
    func(PW,[Client],Skey,[PRIVATE,CRYPTO]),
    func(CP,[Server,Client],Skey,[PRIVATE,CRYPTO])
  ]),

  axioms([
    eqn(PW(C),CP(S,C))
  ])
),

protocol(EKE,
  decls([
    imports([Agent]),
    var([A],Client,[]),
    var([B],Server,[]),
    var([Ra,Rb],Skey,[FRESH,CRYPTO]),
    var([Na,Nb],Nonce,[CRYPTO]),
    var([PA,PB],Pval,[]),
    var([K],Skey,[])
  ]),

  asserts([
    holds(A,[B])
  ]),

  phrases([
    phrase(
      [],
      msg(1,A,B,
```

```
        [A,lowe(crypt(PW(A),kap(Ra)),crypt(CP(B,A),PA))]),
      [eqn(K,kas(PA,Rb))]
    ),
    phrase(
      [],
      msg(2,B,A,
        [B,lowe(crypt(CP(B,A),kap(Rb)),crypt(PW(A),PB)),
        crypt(lowe(K,kas(PB,Ra)),Nb)]),
      [eqn(K,kas(PB,Ra))]
    ),
    phrase([],msg(3,A,B,[A,crypt(K,cat(Na,Nb))]),[]),
    phrase([],msg(4,B,A,[B,crypt(K,Na)]),[])
  ]),

  goals([
    secret(K),
    believes(A,holds(B,[K])),
    believes(B,holds(A,[K]))
  ])
),

environment(Test1,
  decls([
    imports([EKE]),
    const([Alice],Client),
    const([Bob],Server),
    const([Pa],Skey)
  ]),

  axioms([
    eqn(Pa,PW(Alice))
  ]),

  sessions([session(S12,[eqn(A,Alice),eqn(B,Bob)])]),

  exposed([Pa]),

  order(allpar)
```

```
)
)
```

The parser should generate error messages when the specification is not in accord with the concrete syntax. The abstract version of the specification differs from the concrete version by substituting explicit empty lists in places where the specification omits an optional section.

3.3 CIL Abstract Syntax

A context-free grammar for the CIL abstract syntax is given in Appendix C.

A CIL specification has the following parts:

1. declarations (symbol table and slot table)
2. axioms (collected from typespecs, including prelude, and environment)
3. assumptions (localized)
4. rules for state transitions
5. goals (localized)
6. environment

The symbol table lists the declared types of variables and the type signatures of functions. The slot table relates protocol variables to state components; it is explained in Chapter 4 because it is a byproduct of rewrite rule production.

Assertions are either axioms or assertions about protocol variables. Axioms from typespecs and environment specifications are consolidated into a single list as part of the minor transformations. Assertions between protocol variables may be either assumptions or goals to be proved, and they are *localized* by associating them with states. Localization of assertions is discussed in Chapter 4 along with state transitions.

Eventually, goals should be translated into proof objectives that are tied closely to the multiset model, for example, by converting them to testable invariants. For now, the abstract syntax of state assertions is carried over without any change other than the localization.

Currently we plan to carry over environment information without change from the CAPSL abstract syntax, except that constant declarations are extracted and incorporated into the single collected set of declarations and that the axioms from the environment specification are consolidated with typespec axioms into a single list.

3.4 Type Checking and the Symbol Table

The type checker makes a pass through the specification to construct a symbol table. Each symbol table entry is of the (abstract) form

`symbol`(*ident, status, arg-types, value-type, properties*)

The status of a symbol is `op`, `var` or `type`. The status `op` is used for functions and constants.

Symbol table entries are constructed from declarations found in type specifications, protocol variable declarations, and environment declarations. If necessary, entries are added for undeclared dummy variables in typespecs. To avoid possible name conflicts, all variables in type specifications are renamed by qualifying them with the typespec identifier.

Given the abstract syntax of EKE, the translator creates the following symbol declarations and axioms:

```
symbol(Client, type, [], Principal, []),
symbol(Server, type, [], Principal, []),
symbol(PW, op, [Client], Skey, [PRIVATE,CRYPTO]),
symbol(CP, op, [Server,Client], SKey, [PRIVATE,CRYPTO]),
symbol(C.Agent, var, [], Client, []),
symbol(S.Agent, var, [], Server, []),
symbol(A, var, [], Client, []),
symbol(B, var, [], Server, []),
symbol(Ra, var, [], Skey, [CRYPTO, FRESH]),
```

```
symbol(Rb, var, [], Skey, [CRYPTO, FRESH]),
symbol(Na, var, [], Nonce, [CRYPTO]),
symbol(Nb, var, [], Nonce, [CRYPTO]),
symbol(PA, var, [], Pval, []),
symbol(PB, var, [], Pval, []),
symbol(K, var, [], Skey, []),
symbol(Alice, op, [], Client, []),
symbol(Bob, op, [], Server, []),
symbol(Pa, op, [], Skey, [EXPOSED]),
symbol(S12, op, [], SessID, []),
symbol('A', op, [], Role, []),
symbol('B', op, [], Role, []),
symbol(SID, var, [], SessID, [])
```

The last three symbols were not declared explicitly in the specification, but rather added automatically by the translator. For each protocol variable of type `Principal`, or any subtype, there is a corresponding constant of type "Role" (such as `'A'` and `'B'`). These roles are used in the CIL rewrite rules. Moreover, a session identifier variable is declared, also for use in the rewrite rules.

The full symbol table includes not only the symbols arising from the specification, but also the symbols declared in the prelude.

Variables used in typespec axioms are renamed because their scope is local to the typespec, and may be used again with different types in other typespecs. They are given qualified names, consisting of the original name followed by the typespec name.

3.5 Minor Transformations

If the type check completes successfully and the symbol table is generated, some additional checks are made, and some changes are made to the parse tree to bring it closer to the CIL abstract syntax.

• Occurrences of the `crypt` function are replaced by either ped or se, depending on the type of the key.

• Calls on the `con` function are made binary using the right-associative convention.

• The Lowe %-terms are checked to see that they occur only as message fields, and they are lifted to a single occurrence at the top level of each such term, with an error message if that is not possible. For example,

```
{A%D, B%C}K
```

would become

```
{A, B}K%{D, C}K.
```

• Axioms from type specifications and from environment specifications are carried over from the abstract syntax representation without change (except for the unique renaming of dummy variables). They are all consolidated into a single combined list. The axioms from the EKE example would look like this:

```
axioms([
  eqn(PW(C.Agent),CP(S.Agent,C.Agent)),
  eqn(Pa, PW(Alice))
]),
```

The list would also include all axioms from the prelude.

• All function and variable declarations are removed, since the information from them has been collected into the symbol table.

The modified parse tree and symbol table are now ready to hand off to the rewrite rule generation stage.

Chapter 4

Rule Generation

4.1 Multiset Rewrite Rules

The CAPSL Intermediate Language (CIL) serves two purposes: to help define the semantics of CAPSL, and to act as an interface through which protocols specified in CAPSL can be analyzed using a variety of tools.

In most of the tools for protocol analysis, state transition rules for protocol processes make use of pattern-matching. For example, if there is a message `B -> A: B, {Na, Nb}PK(A)`, we infer that `A` will accept only messages whose second field is of the form `{Na, Nb}PK(A)`. "Accepting" a message means that `A` will undergo a state transition as a result of receiving it. CIL is designed to take advantage of this pattern-matching concept.

The purpose of CIL is to unambiguously define the meaning of a protocol specification. An earlier suggestion for the CIL semantics [Mil95] used a small set of primitive operations (send, receive, assign, compare) to characterize state transitions. The new approach summarized in this document uses multiset term rewriting rules that permit state changes to be presented more concisely, and in a way that more nearly matches the requirements of analysis tools. This approach was influenced most directly by an analysis example using Maude, by Denker, Meseguer, and Talcott [DMT98], and by Mitchell's linear logic formulation [Mit98], which is being used by Cervesato and Durgin at Stanford.

Transition rules are 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 function values are typed. "Constant" and "variable" have their usual meaning for logical formulas; variables may be instantiated.

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 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. Facts in the multiset are typically ground terms (no variables) when finite-state search tools are used.

The existential quantifier does not have its ordinary meaning. Quantified variables may be instantiated with any fresh constants. This convention has its foundations in linear logic.

Getting back to the example, A's transition could be expressed by the rule

$\texttt{state}('A', 1, S, A, B, N_a), \texttt{msg}(X, A, S, \{N_a, N_b\}PK(A))$
$\longrightarrow \texttt{state}('A', 2, S, A, B, N_a, \{N_a, N_b\}PK(A)).$

This rule has the prior state of A and the message on the left, and the next state of A on the right. There are two different kinds of facts: facts that represent the state of an agent ($\texttt{state}(\ldots)$) and facts regarding messages ($\texttt{msg}(\ldots)$). State facts have at least three arguments:

$$\texttt{state}(\textit{role, state-label, session-id, } \ldots),$$

where the first argument is the role of the agent in the protocol (e.g., 'A', 'B', etc.), the second argument is a number or other label representing the agent's state in the protocol, and the third argument identifies the session.

Further arguments correspond to values held by the agent in the given state. Message facts have at least four arguments:

$$\texttt{msg}(\textit{sender, receiver, session-id, message-field, } \ldots).$$

The fourth and subsequent arguments are the content of the message, Thus, $\texttt{state}('A', 1, S, A, B, N_a)$ says that an agent playing the 'A' role of the protocol in session S is in a state labeled 1 and holds A's and B's addresses as well as a value N_a. (We will often refer to a value of type `Principal` or a subtype as an "address" or "name," since IP addresses and user names often designate such entities.)

The fact $\texttt{msg}(X, A, S, B, \{N_a, N_b\}PK(A))$ is a message sent from principal X to principal A in session S containing B's address and the concatenation of two nonces N_a and N_b encrypted with the public key of A. The X in the sender position of the received message is a new variable. We assume here (like Paulson) that message facts indicate the true sender of the message, but that value is not available to the receiver of the message. In following sections we explain in detail how we derive CIL rewrite rules from a given CAPSL specification.

4.2 Initialization

Rules with an empty left side are interpreted as initialization or fact-generating rules. For each role in the protocol, an initial state fact is generated with initially held variables (the session ID is among those). The rule

$$\longrightarrow \exists S : \texttt{state}('A', 0, S, A, B), \texttt{state}('B', 0, S, B)$$

creates two facts for two processes of the protocol, one in role 'A' and the other in role 'B'. Both processes start out in state 0 for session S. The agent in the 'A' role initially holds its own address as well as the address of its communication partner. The agent for B holds only its own address. The session ID discriminates the state of one session process for a principal A from another in the same role. The session ID is included in messages so that they will be addressed to the session process that expects them.

Note that the same rule could be used to start any number of sessions, with any choice of principals instantiating the variables A and B (or of whatever subtype A and B were declared). CAPSL's `ENVIRONMENT` specification allows a particular choice of sessions to be specified, if the analysis tool requires that.

In the following we describe the process of translating CAPSL to CIL with the help of examples. A formalization of the translation will be the subject of a forthcoming document.

4.3 Rule Generation Overview

The translator transforms the message list into a set of multiset rewrite rules. Each phrase gives rise to several rules. Each state-changing action prior to the message generates a rule affecting the sender of the next message, with its prior state on the left and the modified state on the right, with a new state label. An assignment adds a new argument position, an equation does not. When new argument positions are added, corresponding slot table entries are created as well.

A message generates two rules, one for the sender and one for the receiver. The sender changes state and introduces a message fact, while the receiver tests for a message fact on the left and changes state. Subsequent actions in the phrase are turned into state-changing rules for the receiver.

Finally, any assumptions or intermediate goals found among the actions are localized and added to the appropriate CIL output list.

A major task of the translator is to determine whether the message list of a protocol is implementable. Given a message like `B -> A: {Na, Nb}PK(A)`, the usual questions arise as to whether `A` is capable of recognizing whether a received message is in fact of the required form, and what data from the message is saved by `A` in the next state. An implementability-testing algorithm is presented in detail in Section 4.6, where it is illustrated on the EKE example.

4.4 Rule Notation

In the following we will represent rewrite rules in an ASCII format as 3-tuples:

```
rule([left-side-facts], [∃ variables], [right-side-facts]).
```

Thus, the initialization rule used as an example above would be written

```
rule([], [S], [state('A', 0, S, A, B), state('B', 0, S, B)]).
```

All variables (symbols of "var" status) occurring in rewrite rules are arbitrary or "dummy" variables in the sense that one can rename all the variables in any rewrite rule without changing the meaning of the rule. Thus, it is mildly misleading to use protocol variables in these positions. We do so because it is mnemonically suggestive and because we can make use of the existing type declarations for these variables.

4.5 Slot Table

A slot table entry relates a protocol variable or a term to an argument position in the state predicate of an agent. It is necessary to store this information because variables in rewrite rules, being dummy variables, can be renamed. Thus, the variables in the state predicate of a process do not necessarily correspond to the protocol variables. To determine whether an assertion about protocol variables, like `HOLDS A: K`, is met by the protocol, one needs detailed information in which position of A's state predicate K is stored (if at all). This information is stored in slot table entries.

Each slot entry is of the (abstract) form

> `slot`(*term, role, argument-position*).

The slot table entries included in the final CIL output will mainly be for variables. Some complex terms may be necessary to store message fields that are not reproducible. This is explained in Section 4.6.

A slot entry `slot(A, 'A', 1)` means that the value of protocol variable A that is held by the agent in role `'A'` may be found in argument $4 = 1 + 3$ of its state predicate. The first three arguments of a state predicate, *role, state-label,* and *session-id,* are not counted because they are never mentioned in assertions.

Thus, `slot(B, 'A', 2)` together with the state fact `state('A', 0, s234, Alice, Bob)` means that the agent playing role `'A'` in session `s234`, presently in state 0, holds `Bob` as the value of its local copy of protocol variable B.

The slot entries are not qualified by state label, because when the state changes, we will only add new slots to the end of the argument list; we will not change or move any of the existing ones. The completed slot table will include slot positions that do not occur in states prior to the one in which the slot was added.

The state predicates in the above fact-generating rule imply the following slot entries:

```
slot(A, 'A', 1),
slot(B, 'A', 2),
slot(B, 'B', 1).
```

4.6 Implementability Algorithm

The implementability algorithm determines what an agent must do to send or receive a message, 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 byproduct of the implementability algorithm.

The implementability algorithm handles actions as well as messages. For each action there is an *acting agent*, namely, the sender of the following message or the receiver of the preceding message in the same phrase. Those agents are identified from the message header: its source, destination, and session ID.

The key concepts for implementability are *computability, receivability,* and *invertibility.* A term is computable by an agent if the necessary variables (or larger subterms) are held (by virtue of appearing in a state fact). A term representing a message field is receivable if the agent holds enough variables and subterms to perform the implied pattern match. A term is invertible if it is an encryption or concatenation from which the agent can extract its component data.

Equational Actions

The right side of an equational action is always tested for computability, as we will describe later. Depending on the computability of the left side, the action is understood to be an assignment, a test for equality, or a combination of both.

If the left side of the action is a variable that is not held by the acting agent, then the action is an assignment. An agent holds a variable if a corresponding slot entry exists. If it is not held, a new slot table entry is created. If the left side of the equation is a variable that is held by an agent, then the equality is interpreted as a statement.

If the left side of the equality is a term using `cat` or `con` and it is not computable, the action is split into a sequence of equalities, one for each component of the concatenation. Each one is then interpreted separately.

Messages

Each message gives rise to two views: the sender's view and the receiver's view, due to the use of the Lowe %-notation. The sender's view is used to test the implementability of the sender's state transition rule, and the receiver's for the receiving rule.

For the sender, the translator must determine whether the sender possesses all fields necessary to compute the message (including the name of the addressee). The *computability* function is called for each field in the fieldlist. The message is implementable only if all fields are computable. The implementability test may have the side effect of inserting additional actions for the sender prior to the message transmission rule, so as to perform necessary calculations using `DENOTES` equations.

For the receiver, the translator must determine whether the receiver can pattern-match the message. For this purpose the *receivability* function is called. A message field is receivable if it is computable or invertible.

Computability:
A term is computable if (1) the agent has the term already in its state (or "memory") or if (2) the agent can recompute the term. Case (1) is fulfilled if (a) a slot for the agent with the term in question is defined (or any other term to which the term in question can be reduced using the axioms), or (b) a denote declaration is given for this term and the right side of the denote declaration is computable for the agent, or (c) the term is defined as a fresh variable that has not yet been created by any other agent and, thus, can be created on the spot. A slot table entry is created in this case. In case (2) the outermost function of the term must be accessible for the agent. This means that it is either a public function or it is a private function for the particular agent. If this is the case, then the computability for all arguments is checked recursively.

Receivability:
To determine whether a term is receivable for an agent the translator first determines whether the term is (1) computable for the agent (using the computability function from above). If this is the case, the test is successful. If this is not the case, then a test determines whether (2) the received term is a variable for which a new slot can be created. If this is the case, the test is successful and the receiver's state is extended. Otherwise, (3) the term is tested for *invertibility*.

Invertibility:
Whether a function is invertible or not depends on the kind of function. For the first implementation of the translator we deal only with the invertible operators `se`, `sd`, `sed`, `pe`, `pd`, `ped`, `con`, and `cat`. We consider all other functions as not being invertible. Later versions of the CAPSL translator will also support user-defined, invertible operators.

Depending on the type of the invertible function, there might be side effects on the memory of an agent. These effects occur because some terms are invertible but not recomputable by an agent. Those terms must be stored to be available for resending in a later protocol step.

We already stated that the meaning of invertibility depends on the kind of operator. Six of the invertible operators are encryption operators. All of them require a key and a field as arguments. An encrypted term `crypt(K,F)` is invertible by an agent, if the agent can compute the inverse key to K and if the field F is receivable. Axioms of type specifications may be used to determine whether an inverse key is computable.

In the case of symmetric-key encryption (`se`, `sd`, `sed`), invertibility of a key corresponds to computability of that key. Given a public-key encryption operator (`pe`, `pd`, `ped`) and a public key K, a term using K is invertible if there is an axiom or assumption of the form `keypair(J, K)` and J is computable. This test is complicated by the fact that J and K may be function calls, the `keypair` axiom may have to be instantiated with a pattern match, and equality of keys may involve reduction modulo other axioms.

Another problem is that the order of fields in the message does not necessarily match the order in which the the receiver must assimilate them. For example, given the message `A -> B : {X}K, K`, and assuming that K is a symmetric key that is not yet held by B, then B cannot decrypt the first field `{X}K` until B has received the second field K. For this reason, the translator may have to make several passes over the list of message fields before it knows whether or not a particular field is invertible.

Encryption with the secret key of a public-key pair yields a term that is invertible but not recomputable by an agent that does not hold the secret key. Thus, receiving a term of this kind has a side effect on the state of the agent. Such a term is stored to make sure that it is available for later protocol steps. For instance, given the situation where agent `B` receives from `A` the term `{A}SK(A)`, assume that `SK` is the private function `SK : Agent -> PKey` and there exists a public function `PK : Agent -> PKey` that delivers inverse keys (i.e., `keypair(PK(A),SK(A))`. Then `B` can decrypt this term using the public-key function and can learn `A`'s name. But `B` cannot recompute this term because of the lack of `A`'s private key. Thus, to enable `B` to forward this term in a later message, it must be stored.

The treatment of `cat` and `con` is a bit different from how encryption operators are handled, because the invertibility of `cat` and `con` does not depend on the computability of one of their arguments. By definition, `con` is always invertible. The `cat` operator is invertible as long as the first argument is of an atomic type, such as `Principal`, `Nonce`, `SKey`, or `PKey`. If both arguments are nonatomic—for example, both arguments are of type `Field`—then `cat` is not invertible.

The first version of the CAPSL translator will not be capable of deriving from a (user-defined) typespec whether or not a new type is atomic. The question of deriving from typespec definitions and axioms whether or not a type is atomic is subject to future work.

Implementability Test – EKE example

We explain how the translator checks the implementability of a message or action in a phrase with the help of the EKE protocol. The first message in EKE is

```
1.  A -> B: A, {kap(Ra)}PW(A)%{PA}CP(B, A).
```

Two rules are generated from this message: one for the sender's view and one for the receiver's view.

The sender's view of this message, `A -> B: A, {kap(Ra)}PW(A)`, requires that `A` knows `B`'s name and its own name, and that `A` can compute the term `{kap(Ra)}PW(A)` to send out the first message. The translator determines whether `A` has all necessary fields to construct this message. The addresses are part of `A`'s initial state (`state('A', 0, S, A, B)`), and the corresponding slot declarations express that `A` indeed has values for these terms. The translator also determines whether `A` is capable of computing the term `{kap(Ra)}PW(A)`. The operator `kap` is predefined in the prelude, and not private. Thus, `kap(Ra)` is accessible for agent `A` if `Ra` is held by it. Since `Ra` is declared as `FRESH`, and has not yet been created by any other agent, it can be newly generated by `A`. `PW` is defined as a private function for clients and, therefore, `PW(A)` is computable for `A`. To summarize, agent `A` can perform all necessary operations to send the first message to `B`.

Since the outcome of the implementability analysis of sending this message is positive, the translator generates a rewrite rule expressing that agent `A` performs a state transition from its latest state to a new state and the first message is sent:

```
rule([state('A',0,SID,A,B)],
     [Ra],
     [state('A',1,SID,A,B,Ra),
```

```
    msg(A,B,SID,A,se(PW(A),kap(Ra)))]).
```

A stores the newly created value Ra in its new state. This fact is also reflected in the corresponding slot entry: slot(Ra, 'A', 3).

The receiver's view of the first message is X -> B: A, {PA}CP(B, A). Note that the X in the sender position of the received message is a new variable. We follow Paulson's approach in that message facts indicate the true sender of the message, but that value is not available to the receiver of the message. We need to define this variable in the symbol table:

```
symbol(X, var, [], Principal, []).
```

B expects to receive a message containing two fields: (1) A's name, and (2) the term {PA}CP(B,A). The translator must determine whether B can pattern-match those fields. For the first message field A, the translator calls the computability test, which fails. But since A is a variable not yet known by B, the translator deduces that a new slot must be created. Thus, the slot slot(A,'B',2) is produced.

The second message field {PA}CP(B,A) has a symmetric-key encryption operation as the outermost function. The translator first tests for computability of this term. This test fails, though B can access the public operator se as well as the key CP(B,A), since B does not have the field PA needed to recompute the term. The second step in the receivability algorithm also fails since {PA}CP(B,A) is not a variable. Invertibility is tested next. The outermost operation se is an invertible operation with the first argument the symmetric key CP(B,A). The translator deduces that B can compute the inverse key, which in this case is CP(B,A) itself. Therefore, the receivability function is called with the second argument of the encryption operator PA. Since this is a variable, a new slot slot(PA,'B',3) is defined.

To summarize, the message X -> B: A, {PA}CP(B, A) is receivable by agent B and the translator produces the following rule and declarations:

```
rule([state('B',0,SID,B), msg(X,B,SID,A,se(CP(B,A),PA))],
      [],
      [state('B',1,SID,B,A,PA)]),
slot(A, 'B', 2),
slot(PA, 'B', 3)
```

The next definition in the message list is an action to be performed by `B`. The translator determines that `K = kas(PA,Rb)` is an assignment, since `B` does not hold `K` yet. Moreover, whether `kas(PA,Rb)` can be computed by the agent must be tested. In our example, `kas` is predefined and the translator determines whether `B` can compute both the terms to which `kas` is applied, namely, `Rb` and `PA`. `Rb` can be generated on the fly because of its freshness property. To find out whether `B` is in possession of `PA` the translator can search for a slot entry having `PA` in its first argument. Because of the slot `slot(PA, 'B', 4)` one can derive that `B` can perform the necessary action and assign a value to `K`. In summary, two slot entries and a rewrite rule are the result of processing this action.

```
rule([state('B',1,SID,B,A,PA)],
     [Rb],
     [state('B',2,SID,B,A,PA,Rb,kas(PA,Rb))]),
slot(Rb, 'B', 4),
slot(K, 'B', 5)
```

4.7 Local Assertions

When messages and actions are converted to state transition rules and assertions are moved into a separate list, the temporal interleaving of intermediate goals or idealization assumptions with the message list must be replaced by a different kind of interleaving, which associates them with network states. A network state is represented with a list of roles and state labels.

For the sake of uniformity, initial assumptions are localized to the network state in which all roles are at state zero. Assertions in the `GOALS` section are localized to the network state in which all roles are in the last states produced by the rule generation process. (Or all such last states, if branching occurs.)

A local assertion is of the (abstract) form

`loc`(*node sequence, assertion*)

where nodes have the following (abstract) syntax:

`loc` (*role, state-label*)

Coming back to our running example, the EKE protocol, we only have to deal with the assumption `HOLDS A: B` and the goals that have to be satisfied at the end of the protocol:

```
  SECRET K;
  BELIEVES A: HOLDS B: K;
  BELIEVES B: HOLDS A: K;
```

Since both agents are in state 5 at the end of the protocol, these labels are taken for the localized goal assertions. Thus, we get

```
assums([
  loc([node(A, 0), node(B, 0)], holds(B, K))
])
goals([
  loc([node(A,5),node(B,5)], secret(K,[])),
  loc([node(A,5),node(B,5)], believes(A,[holds(B,[K])])),
  loc([node(A,5),node(B,5)], believes(B,[holds(A,[K])]))
])
```

We provide local assertions as a uniform way to treat initial assertions and (intermediate) goals. A local assertion must hold for all roles given in the list of localities at the state identified by the corresponding state label. This information will be useful to verify whether or not goals are met by the protocol. In the first implementation of the CAPSL translator we will not provide any further analysis on the basis of this information. Later implementations are expected to do more with this information, such as generating invariants, failure patterns, and attacker rules, or to verify (some of) the goals.

4.8 CIL Specification of EKE

The following CIL representation of the EKE example is a linear (with line breaks and indentation for readability) encoding of the parse tree. Comments about optimization are explained in the Section 4.9.

```
specification(

decls([
  symbol(Client, type, [], Principal, []),
  symbol(Server, type, [], Principal, []),
  symbol(PW, op, [Client], Skey, [PRIVATE,CRYPTO]),
  symbol(CP, op, [Server,Client], SKey, [PRIVATE,CRYPTO]),
  symbol(A, var, [], Client, []),
  symbol(B, var, [], Server, []),
  symbol(Ra, var, [], Skey, [CRYPTO, FRESH]),
  symbol(Rb, var, [], Skey, [CRYPTO, FRESH]),
  symbol(Na, var, [], Nonce, [CRYPTO]),
  symbol(Nb, var, [], Nonce, [CRYPTO]),
  symbol(PA, var, [], Pval, []),
  symbol(PB, var, [], Pval, []),
  symbol(K, var, [], Skey, []),
  symbol(Alice, op, [], Client, []),
  symbol(Bob, op, [], Server, []),
  symbol(Pa, op, [], Skey, []),
  symbol(S12, op, [], SessID, []),
  symbol(C.Agent, var, [], Client, []),
  symbol(S.Agent, var, [], Server, []),
  symbol('A', op, [], Role, []),
  symbol('B', op, [], Role, []),
  symbol(SID, var, [], SessID, []),
  symbol(X, var, [], Principal, []),
  slot(A, 'A', 1),
  slot(B, 'A', 2),
  slot(B, 'B', 1),
  slot(Ra, 'A', 3),
  slot(A, 'B', 2),
  slot(PA, 'B', 3),
  slot(Rb, 'B', 4),
  slot(K, 'B', 5),
  slot(Nb, 'B', 6),
  slot(PB, 'A', 4),
  slot(Nb, 'A', 5),
  slot(K, 'A', 6),
```

```
  slot(Na, 'A', 7),
  slot(Na, 'B', 7)
]),

axioms([
  eqn(PW(C.Agent),CP(S.Agent,C.Agent)),
  eqn(Pa, PW(Alice))
]),

assums([
  loc([node(A, 0), node(B, 0)], holds(B, K))
]),

rules([
  rule([],
       [SID],
       [state('A',0,SID,A,B), state('B',0,SID,B)]),

  rule([state('A',0,SID,A,B)],
       [Ra],
       [state('A',1,SID,A,B,Ra),
       msg(A,B,SID,A,se(PW(A),kap(Ra)))]),
       /* send message 1 */

  rule([state('B',0,SID,B), msg(X,B,SID,A,se(CP(B,A),PA))],
       [],
       [state('B',1,SID,B,A,PA)]),
       /* receive message 1 */

  /* optimize: combine previous and next rule */

  rule([state('B',1,SID,B,A,PA)],
       [Rb],
       [state('B',2,SID,B,A,PA,Rb,kas(PA,Rb))]),

  /* optimize: combine previous and next rule */

  rule([state('B',2,SID,B,A,PA,Rb,K)],
```

```
      [Nb],
      [state('B',3,SID,B,A,PA,Rb,K,Nb),
       msg(B,A,SID,se(CP(B,A),kap(Rb)),se(K,Nb))]),
      /* send message 2 */

rule([state('A',1,SID,A,B,Ra),
       msg(X,A,SID,se(PW(A),PB),se(kas(PB,Ra),Nb))],
      [],
      [state('A',2,SID,A,B,Ra,PB,Nb)]),
      /* receive message 2 */

/* optimize: combine previous and next rule */

rule([state('A',2,SID,A,B,Ra,PB,Nb)],
      [],
      [state('A',3,SID,A,B,Ra,PB,Nb,kas(PB,Ra))]),

/* optimize: combine previous and next rule */

rule([state('A',3,SID,A,B,Ra,PB,Nb,K)],
      [Na],
      [state('A',4,SID,A,B,Ra,PB,Nb,K,Na),
       msg(A,B,SID,se(K,cat(Na,Nb)))]),
      /* send message 3 */

rule([state('B',3,SID,B,A,PA,Rb,K,Nb),
       msg(X,B,SID,se(K,cat(Na,Nb)))],
      [],
      [state('B',4,SID,B,A,PA,Rb,K,Nb,Na)]),
      /* receive message 3 */

/* optimize: combine previous and next rule */

rule([state('B',4,SID,B,A,PA,Rb,K,Nb,Na)],
      [],
      [state('B',5,SID,B,A,PA,Rb,K,Nb,Na),
       msg(B,A,SID,se(K,Na))]),
      /* send message 4 */
```

```
  rule([state('A',4,SID,A,B,Ra,PB,Nb,K,Na),
        msg(X,A,SID,se(K,Na))],
       [],
       [state('A',5,SID,A,B,Ra,PB,Nb,K,Na)])
]),

goals([
  loc([node(A,5),node(B,5)], secret(K,[])),
  loc([node(A,5),node(B,5)], believes(A,[holds(B,[K])])),
  loc([node(A,5),node(B,5)], believes(B,[holds(A,[K])]))
]),

environment(
  Test1,
  [session(S12, [eqn(A,Alice), eqn(B,Bob)])],
  exposed([Pa]),
  order(allpar)
)
```

4.9 Optimization of Rewrite Rules

A rule is created for each action of an agent, and two rules are created for each message. It is possible to perform some optimization by merging some rules. Optimization of the rule set is of great interest when it comes to verification or model-checking issues. State spaces are smaller, and fewer transition rules are available. This way, the performance of a finite-state exploration tool can be increased enormously.

Two rules can be merged if the left side of the second rule can be unified with a subset of the right side of the first rule. The merged rule shows the net effect of applying the two rules in sequence. The original rules can then be dropped, given an argument that there is no other way for the second rule to be applied.

In the EKE example the third, fourth, and fifth rules are subsequent rules of agent `B`. They can be collapsed to one rule.

```
  rule([state('B',0,SID,B), msg(X,B,SID,A,se(CP(B,A),PA))],
       [Rb,Nb],
       [state('B',3,SID,B,A,PA,Rb,kas(PA,Rb),Nb),
        msg(B,A,SID,se(CP(B,A),kap(Rb)),se(kas(PA,Rb),Nb))]),
       /* receive message 1, action, send message 2 */
```

Maximal optimization on the EKE rule list results in the following set of rules:

```
rules([
  rule([],
       [SID],
       [state('A',0,SID,A,B), state('B',0,SID,B)]),

  rule([state('A',0,SID,A,B)],
       [Ra],
       [state('A',1,SID,A,B,Ra),
       msg(A,B,SID,A,se(PW(A),kap(Ra)))]),
       /* send message 1 */

  rule([state('B',0,SID,B), msg(X,B,SID,A,se(CP(B,A),PA))],
       [Rb,Nb],
       [state('B',3,SID,B,A,PA,Rb,kas(PA,Rb),Nb),
          msg(B,A,SID,se(CP(B,A),kap(Rb)),se(kas(PA,Rb),Nb))]),
       /* receive message 1, action, send message 2 */

  rule([state('A',1,SID,A,B,Ra),
        msg(X,A,SID,se(PW(A),PB),se(kas(PB,Ra),Nb))],
       [Na],
       [state('A',4,SID,A,B,Ra,PB,Nb,kas(PB,Ra),Na),
        msg(A,B,SID,se(kas(PB,Ra),cat(Na,Nb)))]),
        /* receive message 2, action, send message 3 */

  rule([state('B',3,SID,B,A,PA,Rb,K,Nb),
        msg(X,B,SID,se(K,cat(Na,Nb)))],
       [],
       [state('B',5,SID,B,A,PA,Rb,K,Nb,Na),
```

```
         msg(B,A,SID,se(K,Na))]),
        /* receive message 3, send message 4 */

  rule([state('A',4,SID,A,B,Ra,PB,Nb,K,Na),
         msg(X,A,SID,se(K,Na))],
        [],
        [state('A',5,SID,A,B,Ra,PB,Nb,K,Na)])
]),
```

The optimization process for EKE cuts down the original eleven rules to six rules. Moreover, the number of different states for an agent is reduced significantly. Agent `A` has only four states and Agent `B` has only three states compared to the six states each of the agents had in the nonoptimized CIL version.

4.10 An Attacker Model

To represent the possible actions of an intruder in the environment, one could add a fact of the form $H(u)$ meaning that the attacker holds the term u. Initial facts of this form would be derived from an `ENVIRONMENT` specification, along with the usual rules showing how an attacker can learn terms from messages and can compose and decompose terms. Here, we first describe how the initial knowledge of an intruder is determined from a CIL specification. Then, we present general rules for how the intruder learns new fields from those fields he knows.

Initially the intruder holds all exposed constants and terms as declared in the environment section. For instance, in the EKE protocol the exposed constant `Pa` results in a fact-generating rule for the intruder:

```
  rule([],
        [],
        [H(Pa)]) .
```

If a principal is exposed, then all private functions defined for this principal are also exposed. Assume the environment specification

```
ENVIRONMENT Trouble;
  IMPORTS EKE;
  CONSTANTS Cecile: Server, EXPOSED;
  END;
```

The corresponding CIL specification has the following symbol entry:

```
symbol(Cecile, op, [], Server, [EXPOSED]).
```

The intruder would not just know the name 'Cecile', but could also access the 'Cecile'-row of all private functions of servers. For instance, `CP(Cecile, X)` for any client X is compromised. Thus, we would have the attacker rule

```
  rule([],
       [],
       [H(Cecile), H(CP(Cecile, X)])
```

Rules must be adapted to express that an intruder can guess some of the protocol variables. Protocol variables that are initially held by an agent and that are not declared as being `CRYPTO` are part of the attacker's knowledge. Since all the variables in the initial rule for EKE are not `CRYPTO`, we can add the rule

```
  rule([state('A', 0, SID, A, B), state('B', 0, SID, B)],
       [],
       [state('A', 0, SID, A, B), state('B', 0, SID, B),
        H(A), H(SID), H(B)]) .
```

Note that reference to the initial state facts is necessary in putting variables in the rule into context. In other words, just adding a rule `rule([], [SID], [H(A), H(SID), H(B)])` does not work, since all rule variables are dummy variables. This rule would express that the intruder knows any session ID and any principal name. But we want to express something different: Given a specific initial state for the EKE protocol with agents A and B and a session ID, then the intruder knows those same names and session ID. This is achieved by incorporating the agent facts into the attacker rule.

Nonces have the special property that they may be created on the fly. Thus, they do not need to be part of the initial state of any of the protocol agents. Nevertheless, if they are not declared to be `CRYPTO` the intruder might guess them. Therefore, given a nonce declaration that is not `CRYPTO` such as

```
VARIABLES N: Nonce;
```

and the corresponding rule in which this nonce is created, for example,

```
rule([state('A', 2, SID, ...)],
     [N],
     [state('A', 3, SID, ..., N)])
```

we need to add the rule

```
rule([state('A', 3, SID, ..., N)],
     [],
     [state('A', 3, SID, ..., N), H(N)]).
```

Moreover, the intruder can access all typespec functions that are not `PRIVATE`. For example, we would have

```
rule([H(X), H(K)],
     [],
     [H(X), H(K), H(se(K,X))]
```

and

```
rule([H(X), H(Y)],
     [],
     [H(X), H(Y), H(cat(X,Y))]
```

Similar rules would be derived for each defined typespec function.

Axioms from typespecs like

```
sd(K, se(K, X)) = X
```

are carried over to the CIL output. They could, in principle, be used as reduction rules to be applied in conjunction with the multiset rewriting. However, the equational axioms supplied might not be in a convenient (directed, convergent) form for use as reduction rules. For the built-in operators a more efficient tactic would be to include specific attacker rules. For instance, axioms for the built-in encryption operators express their invertibility. Those could be made explicit in rules like

```
rule([H(se(K,X)), H(K)],
     [],
     [H(se(K,X)), H(K), H(X)])
```

where K is of type Skey.

The behavioral capability of an intruder must be defined with the help of rules. Taking the Dolev-Yao approach [DY83], one could introduce rules for overhearing, intercepting, faking, and replaying messages.

Generally, the intruder can learn all fields from a transmitted message. For each transmitted message there is a rule like

```
rule([msg(A, B, SID, A, U, V)],
     [],
     [H(A), H(B), H(SID), H(U), H(V)]).
```

Note that since the variables are universally quantified, this rule actually applies to all possible messages with the same length and type signature. We need one rule of this kind for each message format that appears in the protocol.

This rule destroys the original message, but the message can be recreated by the attacker using a rule for faking or replaying messages.

For messages with one content field, we have

```
rule([H(A), H(B), H(F), H(SID)],
     [],
     [H(A), H(B), H(F), H(SID), msg(A,B,SID,F)])
```

We expect that the CAPSL translator (but perhaps not the first version) will produce rules of this kind. They would serve primarily to explain the assumptions made about the attacker, and would probably not be practical to use directly as an attacker model for a finite-state search tool.

Chapter 5

Concluding Remarks and Future Work

We have driven the ball as far as we can with a single swing. Now we have to catch up to where it lies and hit it again. The most immediately useful product of a CAPSL compiler is the set of rewrite rules expressing the state transitions of a protocol. We have a good idea of what the rewrite rules should look like, and we have produced some examples and some informal principles to guide us.

We need to implement what we have envisioned so far, both as a firm semantics and as executable code. This will give us the ability to parse and typecheck CAPSL specifications, and to produce the rewrite rules. The next drive will tell us how to deal with goals and the attacker model.

We don't understand enough yet about the semantics of goals. It is very likely, however, that they are all expressible in the multiset model, and that that they are closely tied to the attacker model.

There are other directions we also intend to explore, to expand the CAPSL and CIL concepts to their full generality. There should be a universal way to handle invertibility of operators and other mechanisms for which we have temporarily made limiting simplifying assumptions. We do not yet fully understand some of the concepts implied by declared properties such as freshness and unguessability.

There will also be plenty of work to do to make the software tools practical, both in software engineering areas such as the choice of language, and in algorithmic areas such as optimization of rewrite rules.

Bibliography

[BAN90] M. Burrows, M. Abadi, and R. Needham. A logic of authentication. *ACM Transactions on Computer Systems*, 8(1):18–36, 1990.

[BM93] S. M. Bellovin and M. Merritt. Augmented Encrypted Key Exchange: a Password-Based Protocol Secure Against Dictionary Attacks and Password File Compromise. In *1st ACM Conf. on Computer and Communications Security*, pages 244–250. ACM SIGSAC, 1993.

[BMM99] S. Brackin, C. Meadows, and J. Millen. CAPSL Interface for the NRL Protocol Analyzer. In *IEEE Symposium on Application-Specific Systems and Software Engineering Technology, ASSET '99, Dallas* , 1999. *To appear.*

[Bra97] S. Brackin. An interface specification language for automatically analyzing cryptographic protocols. In *Symposium on Network and Distributed System Security*. Internet Society, February 1997.

[Car94] U. Carlsen. Generating formal cryptographic protocol specifications. In *IEEE Symposium on Research in Security and Privacy*, pages 137–146. IEEE Computer Society, 1994.

[DMT98] G. Denker, J. Meseguer, and C. Talcott. Protocol Specification and Analysis in Maude. In N. Heintze and J. Wing, editors, *Proc. of Workshop on Formal Methods and Security Protocols, 25 June 1998, Indianapolis, Indiana*, 1998. `http://www.cs.bell-labs.com/who/nch/fmsp/index.html`.

[DY83] D. Dolev and A. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, IT-29:198–208, 1983. Also STAN-CS-81-854, May 1981, Stanford U.

[Kem89] R. Kemmerer. Analyzing encryption protocols using formal verification techniques. *IEEE Journal on Selected Areas in Communication*, 7(4), May 1989.

[Low96] G. Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In *Proceedings of TACAS*, volume 1055 of *Lecture Notes in Computer Science*, pages 147–166. Springer-Verlag, 1996.

[Low98] G. Lowe. Casper: a compiler for the analysis of security protocols. *Journal of Computer Security*, 6(1):53–84, 1998.

[MCF87] J. Millen, S. Clark, and S. Freedman. The Interrogator: protocol security analysis. *IEEE Transactions on Software Engineering*, SE-13(2):274–288, February 1987.

[Mea91] C. Meadows. A system for the specification and verification of key management protocols. In *IEEE Symposium on Security and Privacy*, pages 182–195. IEEE Computer Society, 1991.

[Mil95] J. Millen. The Interrogator model. In *IEEE Symposium on Security and Privacy*, pages 251–260. IEEE Computer Society, 1995.

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

[Mit98] J. Mitchell. Invited Talk, CAV'98, slides, 1998. `http://theory.stanford.edu/people/jcm/home.html`.

[Ros95] A. W. Roscoe. Modelling and verifying key-exchange protocols using CSP and FDR. In *8th IEEE Computer Security Foundations Workshop*, pages 98–107. IEEE Computer Society, 1995.

Appendix A

CAPSL Concrete Syntax

reserved words

TYPESPEC	PROTOCOL	IMPORTS	VARIABLES	CONSTANTS
DENOTES	TYPES	ASSUMPTIONS	HOLDS	BELIEVES
KNOWS	SECRET	SESSION_SECRET	MESSAGES	INCLUDE
GOALS	END	AXIOMS	FUNCTIONS	ENVIRONMENT
SESSION	EXPOSED	CRYPTO	RANDOM	FRESH
PRIVATE	IF	THEN	ELSE	ENDIF
ORDER	ASSUME	PROVE		

predefined names

Principal	Field	SKey	PKey	Nonce	Timestamp	Pval
se	sd	sed	sha	pe	pd	ped
xor	kap	kas	seal	verify	cat	first
rest	con	head	tail	later	near	keypair
atomic						

meta rules

Optional constructs are shown between square brackets, as [<*x*>].

Constructs enclosed in curly brackets, as {<*x*>;}, may be repeated any positive number of times.

Brackets occurring as terminal symbols in the grammar should be recognizable because they are in bold face.

There is a general rule for forming nonterminals representing lists of items separated by commas:

<*x_list*> ::= <*x*> | <*x*> **,** <*x_list*>

identifiers

<*ident*> ::= {<*character*>}

<*character*> ::= **A** | ... | **Z** | **a** | ... | **z** | **_** | **0** | **1** | ... | **9**

<*number*> ::= {<*digit*>}

<*digit*> ::= **0** | **1** | ... | **9**

<*property*> ::= **CRYPTO** | **FRESH** | **RANDOM** | **PRIVATE** | **EXPOSED**

terms

<*term*> ::= <*functioncall*> | <*ident*> | <*bracket*> | <*lowe*>

<*functioncall*> ::= <*ident*> **(**<*term_list*>**)**

<*bracket*> ::= { <*term_list*> } ['] [<*term*>] | **[** <*term_list*> **]** ['] [<*term*>]

<*lowe*> ::= <*term*> **%** <*term*>

It would be possible to write the grammar in such a way as to restrict the appearance of the Lowe %-operator to protocol sections and subterms where it makes sense. That was not done here, so that the grammar would be simpler. The necessary checks can be done easily later in the translation.

phrases

<*phrases*>	::=	<*phrase*> [/] [<*phrases*>]
<*phrase*>	::=	[{<*action*>;}] <*message*> [{<*action*>;}] | <*invocation*> | <*selection*>
<*action*>	::=	<*statement*> | **ASSUME** <*assertion*> | **PROVE** <*assertion*>
<*message*>	::=	[<*number*>.] <*ident*> -> <*ident*> **:** <*term_list*>**;**
<*invocation*>	::=	**INCLUDE** <*ident*>**;**
<*selection*>	::=	**IF** <*statement*> **THEN** <*phrase*> **ELSE** <*phrase*> **ENDIF;**

assertions

<*assertion*>	::=	**HOLDS** <*ident*> **:** <*ident_list*> | **BELIEVES** <*ident*> **:** <*assertion*> | **KNOWS** <*ident*> **:** <*assertion*> | **SECRET** <*ident*> [**:** <*ident_list*>] | **SESSION_SECRET** <*ident*> [**:** <*ident_list*>] | <*statement*>

<statement>	::=	*<equation>* | *<term>* | **IF** *<statement>* **THEN** *<statement>* [**ELSE** *<statement>*] **ENDIF**
<equation>	::=	*<term>* **=** *<term>*

declarations

<declaration>	::=	**TYPES** {*<typedecl>*} | **FUNCTIONS** {*<funcdecl>*} | **VARIABLES** {*<vardecl>*} | **CONSTANTS** {*<vardecl>*} | **DENOTES** {*<denotedecl>*} | **IMPORTS** *<ident_list>*; |
<typedecl>	::=	*<ident_list>* **:** *<ident>*;
<funcdecl>	::=	*<ident>*[**(***<ident_list>***)**] **:** *<ident>* [**,***<property_list>*];
<vardecl>	::=	*<ident_list>* **:** *<ident>* [**,***<property_list>*];
<denotedecl>	::=	*<term>* **=** *<term>*;

order

<order>	::=	*<ident>* | **(***<parallel>***)** | **(***<serial>***)**
<parallel>	::=	*<order>* { || *<order>*}
<serial>	::=	*<order>* { **;** *<order>*}

specifications

<*specification*>	::=	{<*specpart*>}
<*specpart*>	::=	<*typespec*> | <*protocol*> | <*environment*>
<*typespec*>	::=	**TYPESPEC** <*ident*>**;** {<*declaration*>} [**AXIOMS** {<*statement*>**;**}] **END;**
<*protocol*>	::=	**PROTOCOL** <*ident*>**;** {<*declaration*>} [**ASSUMPTIONS** {<*assertion*>**;**}] **MESSAGES** <*phrases*> [**GOALS** {<*assertion*>**;**}] **END;**
<*environment*>	::=	**ENVIRONMENT** <*ident*>**;** {<*declaration*>} [**AXIOMS** {<*statement*>**;**}] {<*session*>} [**EXPOSED** <*term_list*>**;**] [**ORDER** <*order*>**;**] **END;**
<*session*>	::=	**SESSION** <*ident*>**;** {<*equation*>**;**}

Appendix B

CAPSL Abstract Syntax

Notation

We assume the declarations of identifiers and we use the same BNF notation, as in Appendix A, except that we do not use brackets to indicate optional items or repetition, and we have with the following additional meta-rules:

<x_seq> ::= <x> | <x> <x_seq>

<x_seq*> ::= <empty> | <x> <x_seq*>

These rules are interpreted as defining parse trees rather than character strings, so the empty element *¡empty¿* is a particular leaf subtree.

Order of subtrees and leaves is important; the meaning of an identifier, for example–whether it is a variable or type or property–depends on its position in order.

The terminal symbols in this grammar are node names, which correspond roughly to the nonterminals in the concrete syntax.

specifications

<specification> ::= **specification** <component_seq>

<*component*> ::= <*typespec*> | <*protocol*> | <*environment*>

<*typespec*> ::= **typespec** <*ident*> <*decls*> <*axioms*>

<*protocol*> ::= **protocol** <*ident*> <*decls*> <*asserts*>
<*phrase_seq*> <*goals*>

<*environment*> ::= **environment** <*ident*> <*decls*> <*axioms*>
<*sessions*> <*exposed*> <*order*>

<*goals*> ::= **goals** <*assert_seq**>

<*sessions*> ::= **sessions** <*session_seq*>

<*session*> ::= **session** <*ident*> <*equation_seq*>

<*exposed*> ::= **exposed** <*term_seq**>

<*order*> ::= **order** <*orderspec*>

<*orderspec*> ::= <*ident*> | **allpar** |
par <*orderspec*> <*orderspec*> |
seq <*orderspec*> <*orderspec*>

declarations

<*decls*> ::= **decls** <*decl_seq*>

<*decl*> ::= <*typedec*> | <*funcdec*> | <*vardec*> |
<*denote*> | <*import*> | <*constant*>

<*typedec*> ::= **types** <*ident_seq*> <*ident*>

The identifiers above are type names and the supertype. The supertype `Field` will be supplied by default, if missing in the concrete syntax.

<*funcdec*> ::= **func** <*ident*> <*ident_seq**> <*ident*> <*property_seq**>

The identifiers above are the function name, the argument types, the value type, and the properties. An empty argument type list means that the function is really a constant.

<*vardec*>	::=	**var** <*ident_seq*> <*ident*> <*property_seq**>

The identifiers above are a sequence of variable names, their type, and their properties.

<*denote*>	::=	**denote** <*term*> <*term*>
<*import*>	::=	**imports** <*ident_seq*>
<*constant*>	::=	**const** <*ident_seq*> <*ident*> <*property_seq**>

The identifiers above are constant names, followed by their common type and their properties.

assertions

<*axioms*>	::=	**axioms** <*stmt_seq**>
<*asserts*>	::=	**asserts** <*assert_seq**>
<*stmt*>	::=	<*equation*> | <*term*> | <*if*>
<*if*>	::=	**if** <*stmt*> <*stmt*> <*stmt*>
<*assert*>	::=	<*holds*> | <*believes*> | <*knows*> | <*secrets*> | <*sessec*> | <*stmt*>
<*equation*>	::=	**eqn** <*term*> <*term*>
<*holds*>	::=	**holds** <*ident*> <*ident_seq*>
<*believes*>	::=	**believes** <*ident*> <*assert*>
<*knows*>	::=	**knows** <*ident*> <*assert*>
<*secret*>	::=	**secret** <*ident*> <*ident_seq**>
<*sessec*>	::=	**sessec** <*ident*> <*ident_seq**>

phrases

<*phrase*>	::=	**phrase** <*action_seq**> <*message*> <*action_seq**> |
		include <*ident*> |
		select <*stmt*> <*phrase*> <*phrase*>
<*action*>	::=	<*statement*> |
		assume <*assert*> |
		prove <*assert*>
<*message*>	::=	**msg** <*number*> <*ident*> <*ident*> <*term_seq*>

The two identifiers above are the sender and receiver; the message number will be filled in somehow, perhaps as zero, if missing from the concrete syntax.

terms

<*term*>	::=	<*fncall*> | <*ident*> | <*lowe*>
<*fncall*>	::=	<*ident*> <*term_seq*>
<*lowe*>	::=	**lowe** <*term*> <*term*>

Note that the abstract syntax does not restrict the appearance of the Lowe operator.

Appendix C

CIL Abstract Syntax

Notation

We use the same BNF notation as in Appendix B, and we assume the rules for identifiers from Appendix A.

specifications

<*specification*> ::= **specification** <*decls*> <*axioms*> <*assums*> <*rules*> <*goals*>
<*environment*>

Variables in the assumptions, goals, and environment are protocol variables, whereas variables in rules and axioms are logical ("dummy") variables.

environment

<*environment*> ::= **environment** <*ident*> <*session_seq*> <*exposed*> <*order*>

<*session*> ::= **session** <*ident*> <*equation_seq*>

<exposed> ::= **exposed** <term_seq*>

<order> ::= **order** <orderspec>

<orderspec> ::= <ident> |
allpar |
par <orderspec> <orderspec> |
seq <orderspec> <orderspec>

The identifier above is a session identifier.

declarations

<decls> ::= **decls** <decl_seq>

<decl> ::= <symbol> | <slot>

<symbol> ::= **symbol** <ident> <status> <ident_seq*> <ident> <property_seq*>

The identifiers above are the name of the symbol, the argument types, and the value type.

<slot> ::= **slot** <term> <ident> <number>

The identifier above is the agent. The number is the position where the term is stored in the state predicate.

<status> ::= **op** | **var** | **type**

assertions

<axioms> ::= **axioms** <stmt_seq*>

<assums> ::= **assums** <locassert_seq*>

<goals> ::= **goals** <locassert_seq*>

<locassert> ::= **loc** *<node_seq>* *<assert>*

<node> ::= **node** *<ident>* *<number>*

The identifier above is a role, and the number is a state label.

<assert> ::= *<holds>* | *<believes>* | *<knows>* | *<secrets>* | *<sessec>* | *<stmt>*

<stmt> ::= *<equation>* | *<term>* | *<if>*

<if> ::= **if** *<stmt>* *<stmt>* *<stmt>*

<equation> ::= **eqn** *<term>* *<term>*

<holds> ::= **holds** *<ident>* *<ident_seq>*

<believes> ::= **believes** *<ident>* *<assert>*

<knows> ::= **knows** *<ident>* *<assert>*

<secret> ::= **secret** *<ident>* *<ident_seq*>*

<sessec> ::= **sessec** *<ident>* *<ident_seq*>*

rules

<rules> ::= **rules** *<rule_seq>*

<rule> ::= **rule** *<fact_seq*>* *<ident_seq*>* *<fact_seq*>*

<fact> ::= *<message>* | *<state>*

<message> ::= **msg** *<ident>* *<ident>* *<ident>* *<term_seq>*

The three identifiers above are the sender, the receiver and the session id.

<state> ::= **state** *<ident>* *<number>* *<ident>* *<term_seq>*

The two identifiers above are the role of an agent and the session identifier; the state number is automatically filled in as a consecutive number for each protocol agent starting with zero.

terms

<*term*>	::=	<*fncall*> | <*ident*>
<*fncall*>	::=	<*ident*> <*term_seq*>

Appendix D

Predefined Type Specifications

Field

The Field type is the universal supertype for all message fields. All typespecs import this one implicitly, just as this one implicitly imports Boolean and Object. The Object type is used only to give Field and Boolean a supertype.

The subtype *Atom* for fields of a well-defined length is the default supertype for all types that do not have an explicitly declared supertype.

The predicate `atomic` is supposed to be a recognizer for the Atom subtype. Formally, to say this we would need a way to define Atom as the subtype of field whose elements satisfy `atomic`. CAPSL does not (as yet) support explicit subtype constructors, so all we can say formally is that every Atom is atomic.

```
TYPESPEC FIELD;
TYPES
    Field: Object;
    Atom: Field; /* satisfying atomic */
    Nonce, Principal;
    SessID: Nonce;
FUNCTIONS
    cat(Field, Field): Field;
    first(Field): Field;
```

```
    rest(Field): Field;
    atomic(Field): Boolean;
VARIABLES A: Atom; X, Y: Field;
AXIOMS
    atomic(A);
    first(cat(A, X)) = A;
    rest(cat(A, X)) = X;
    cat(X, cat(Y, Z)) = cat(cat(X, Y), Z);
END;
```

Symmetric-Key Encryption

The basic Skey type is used by several sets of encryption operators. The only operator given in this root typespec is a hash function.

The DES system could be modeled with the `se, sd` pair. The single-operator version `sed` is included for completeness, but the only form of it that is commonly seen in practice is the more specific `xor` given below.

```
TYPESPEC SKEY;
TYPES Skey;
FUNCTIONS
    sha(Field): Skey;
END;
```

```
TYPESPEC DSKE;
IMPORTS SKEY;
FUNCTIONS
    se(Skey, Field): Field;
    sd(Skey, Field): Field;
AXIOMS
    sd(K, se(K, D)) = D;
    se(K, sd(K, D)) = D;
    IF atomic(D) THEN atomic(se(K, D)) ENDIF;
    IF atomic(D) THEN atomic(sd(K, D)) ENDIF;
```

```
    END;


TYPESPEC SSKE;
IMPORTS SKEY;
FUNCTIONS
    sed(Skey, Field): Field;
AXIOMS
    sed(K, sed(K, D)) = D;
    IF atomic(D) THEN atomic(sed(K, D)) ENDIF;
END;



TYPESPEC XOR;
IMPORTS SKEY;
FUNCTIONS
    xor(Skey, Skey): Skey;
    0: Skey;
AXIOMS
    xor(X, X) = 0;
    xor(X, 0) = X;
    xor(X, Y) = xor(Y, X);
    xor(X, xor(Y, Z)) = xor(xor(X, Y), Z);
END;
```

Public-Key Encryption

As in symmetric-key encryption, there is a basic public-key type, used for both public and private keys. The single-operator version `ped` models RSA at a very abstract level.

```
TYPESPEC PKEY;
TYPES Pkey;
FUNCTIONS
    keypair(Pkey, Pkey): Boolean;
```

```
AXIOMS
    keypair(J, K) = keypair(K, J);
END;


TYPESPEC SPKE;
IMPORTS PKEY;
FUNCTIONS
    ped(Pkey, Field): Field;
AXIOMS
    IF keypair(J, K) THEN ped(J, ped(K, X)) = X ENDIF;
    IF keypair(J, K) THEN ped(K, ped(J, X)) = X ENDIF;
    IF atomic(X) THEN atomic(ped(K, X)) ENDIF;
END;


TYPESPEC DPKE;
IMPORTS PKEY;
FUNCTIONS
    pe(Pkey, Field): Field;
    pd(Pkey, Field): Field;
AXIOMS
    IF keypair(J, K) THEN pd(J, pe(K, X)) = X ENDIF;
    IF keypair(J, K) THEN pd(K, pe(J, X)) = X ENDIF;
    IF keypair(J, K) THEN pe(J, pd(K, X)) = X ENDIF;
    IF keypair(J, K) THEN pe(K, pd(J, X)) = X ENDIF;
    IF atomic(X) THEN atomic(pe(K, X)) ENDIF;
    IF atomic(X) THEN atomic(pd(K, X)) ENDIF;
END;
```

Key Agreement

This key agreement type is meant to express the basic properties of Diffie-Hellman key agreement. The kap operation creates a public value that can be combined with an Skey to produce another Skey using `kas`. This type specification omits significant relations that emerge when `kap` is implemented by raising a known base value to the Skey power modulo a prime.

```
TYPESPEC KeyAgreement;
IMPORTS SKEY;
TYPES Pval: Skey;
FUNCTIONS
    kap(Skey): Pval;
    kas(Pval, Skey): Skey;
AXIOMS
    kas(kap(Ka),Kb) = kas(kap(Kb),Ka);
END;
```

Public-Key Sealing

The following public-key `seal` operation could be implemented with a keyed hash, but it may also be viewed as a primitive operation. It could be the basis for a signature if one assumes that the sealing key is private to the signer.

```
TYPESPEC PKSeal;
IMPORTS PKEY;
TYPES Pseal;
FUNCTIONS
    seal(Pkey, Field): Pseal;
    verify(Pkey, Pseal, Field): Boolean;
AXIOMS
    IF keypair(J, K) THEN verify(K, seal(J, X), X) ENDIF;
END;
```

Timestamps

```
TYPESPEC TIMESTAMP;
TYPES Timestamp;
FUNCTIONS
```

```
    later(Timestamp,Timestamp): Boolean;
    near(Timestamp,Timestamp): Boolean;
AXIOMS
    later(R, R) = false;
    IF later(R, S) THEN  NOT later(S, R) ENDIF;
    IF later(R, S) AND later(S, T) THEN later(R, T) ENDIF;
    near(R, R);
    IF near(R, S) THEN near(S, R) ENDIF;
END;
```

List

```
TYPESPEC LIST;
TYPES List: Field;
FUNCTIONS
    con(Field, Field): List;
    head(List): Field;
    tail(List): Field;
AXIOMS
        head(con(X, Y)) = X;
        tail(con(X, Y)) = Y;
END;
```

Appendix E

CAPSL Examples

EKE

The basic "encrypted key exchange" protocol, due to Bellovin and Merritt [BM93], might be represented as follows. A typical environment specification is also shown.

```
TYPESPEC Agent;
TYPES Client, Server: Principal;
FUNCTIONS
  PW(Client): Skey, PRIVATE, CRYPTO;
  CP(Server,Client): Skey, PRIVATE, CRYPTO;
AXIOMS
  PW(C) = CP(S, C);
END;

PROTOCOL EKE;
IMPORTS Agent;
VARIABLES
    A: Client;
    B: Server;
    Ra, Rb: Skey, FRESH, CRYPTO;
    Na, Nb: Nonce, CRYPTO;
    PA, PB: Pval;
```

```
    K: Skey;
    T1, T2, T3: Field;
ASSUMPTIONS
    HOLDS A: B;
MESSAGES
    T1 = {kap(Ra)}PW(A);
 1. A -> B: A, T1%{PA}CP(B, A);
    K = kas(PA, Rb);
    T2 = {kap(Rb)}CP(B, A);
    T3 = {Nb}K;
 2. B -> A: B, T2%{PB}PW(A),T3%{Nb}kas(PB, Ra);
    K = kas(PB, Ra);
 3. A -> B: A, {Na, Nb}K;
 4. B -> A: B, {Na}K;
GOALS
  SECRET K;
  BELIEVES A: HOLDS B: K;
  BELIEVES B: HOLDS A: K;
END;

ENVIRONMENT Test1;
IMPORTS EKE;
CONSTANTS
    Alice: Client;
    Bob: Server;
    Pa: Skey;
ASSUMPTIONS
    Pa = PW(Alice);
SESSION S12;
    A = Alice;
    B = Bob;
EXPOSED Pa;
END;
```

Otway-Rees

```
TYPESPEC PAgent;
TYPES Client, Server: Principal;
FUNCTIONS
    Ks(Client): Skey, PRIVATE, CRYPTO;
    SK(Server, Client): Skey, PRIVATE, CRYPTO;
END;

PROTOCOL Otway_Rees;
IMPORTS PAgent;
VARIABLES
    A, B: Client;
    S: Server;
    M: Nonce;
    Na, Nb: Nonce, CRYPTO;
    Kas, Kbs: Skey;
    Kab: Skey, CRYPTO, FRESH;
    T1,T2: Field;
ASSUMPTIONS
    HOLDS A: B, S;
    HOLDS B: S;
MESSAGES
 1. A -> B: M, A, B, {Na, M, A, B}Ks(A)%T1;
 2. B -> S: M, A, B, T1%{Na, M, A, B}SK(S,A), {Nb, M, A, B}SK(S,B);
 3. S -> B: M, {Na, Kab}SK(S,A)%T2, {Nb, Kab}SK(S,B);
 4. B -> A: M, T2%{Na, Kab}Ks(A);
GOALS
    SECRET Kab;
END;
```

SSL Handshake

The Secure Socket Layer (SSL) Handshake Protocol, version 3, is specified in the IETF (Internet Engineering Task Force) document draft-hickman-netscape-ssl-01.txt on IETF FTP sites. The CAPSL example is a partial version that captures only one of the cipher specs options, Diffie-Hellman, and mentions only one other, RSA. This version does not perform client authentication. For simplicity it is assumed that the client offers only a single cipher spec option and the server accepts it.

The CAPSL text illustrates conditional selection of subprotocols and the DENOTES section. The sha operator is used wherever hashes are called for, simplifying out much of the detailed construction of hashes and key material.

```
TYPESPEC SSLAgent;
TYPES Client, Server: Principal;
FUNCTIONS
    Xc(Client): Skey, PRIVATE, CRYPTO;
    Xs(Server): Skey, PRIVATE, CRYPTO;
    Kp(Server): Pkey;
    Ks(Server): Pkey, PRIVATE, CRYPTO;
AXIOMS
    keypair(Kp(S),Ks(S));
END;

PROTOCOL SSL_Handshake;
IMPORTS SSLAgent;
TYPES CipherSpec;
VARIABLES
    C: Client;
    S: Server;
    CH: Nonce, RANDOM;          /* challenge */
    CS: CipherSpec;
    X,Y: Pval;                  /* key agreement public values */
    CID: Nonce;                 /* connection id */
    SID: Nonce;                 /* session id */
    PMK: Skey;                  /* pre-master-key */
    MK: Skey:                   /* master key */
    HH, SHH, Kwc, Kws: Skey;
CONSTANTS
    DH, RSA: CipherSpec;
DENOTES
    PMK = kas(kap(Xs(S)),Xc(C));/* DH negotiated pre-master key */
```

```
    MK = sha({PMK,CH,CID});
    HH = sha({C,S,CH,CS,Y});    /* handshake hash */
    SHH = sha({S,C,CID,SC,CS}); /* server handshake hash */
    Kwc = sha({MK,CH,CID});     /* client write key */
    Kws = sha({MK,CH,CID});     /* server write key */
ASSUMPTIONS
    HOLDS C: S, CS;
MESSAGES
    C -> S: C, CH, CS;           /* ClientHello */
    IF     CS = DH  THEN INCLUDE SSL_DH;
    ELSEIF CS = RSA THEN INCLUDE SSL_RSA;
    ELSE ...
    ENDIF;
GOALS
    SECRET MK;
    SECRET Kwc;
    SECRET Kws;
END;

PROTOCOL SSL_DH;
IMPORTS SSL;
MESSAGES
    X = kap(Xs(S));
 1. S -> C: CID, {X}Kp(S), DH;   /* ServerHello */
    Y = kap(Xc(C));
    PMK = kas(X,Xc(C));
 2. C -> S: Y;            /* ClientKeyExchange */
 3. C -> S: {CID,HH}Kwc;  /* Client-Finished */
    PMK = kas(Y,Xs(S));
 4. S -> C: {SID,SHH}Kws; /* Server-Finished */
END;
```

Secure Remote Password (SRP) Protocol

SRP, a protocol in the EKE family designed to defeat password guessing, was developed at Stanford (http://jafar.stanford.edu/srp/design.html). For simplicity, we make a few modifications.

1. There is no mention of the modulus N for finite field arithmetic. Arithmetic is done on the `SKey` type.

2. The checks that B, u, and A are not zero are omitted.

3. Messages 3 and 4 to confirm reception of the key K are simpler than the suggested ones.

```
TYPESPEC User;
TYPES User, Host: Principal;
FUNCTIONS
    g: Skey;    /* generator */
    p(User): Field, PRIVATE, CRYPTO;  /* password */
    s(Host,User): Field, PRIVATE;     /* salt */
    v(Host,User): Skey, PRIVATE;      /* password verifier */
AXIOMS
    v(H1,U1) = g^sha({s(H1,U1),p(U1)});
END;


PROTOCOL SRP;
IMPORTS User;
VARIABLES
    U: User;
    H: Host;
    A, B: Skey;
    a, b, u: Skey, FRESH, CRYPTO;
    K,S,s,x: Skey;
DENOTES
    A = g^a;
    B = v(H,U) + g^b;
    x = sha({s,p(U)});
    v = g^x;
ASSUMPTIONS
    HOLDS U: H;
MESSAGES
 1. U -> H: U, A;             /* U generates a */
```

```
   S = (A*v(H,U)^u)^b;    /* H generates b */
   K = sha(S);
 2. H -> U: s(H,U)%s, B, u;
   S = (B - v)^(a + u*x);
   K = sha(S);
 3. U -> H: {A}K;            /* proves U holds K */
 4. H -> U: {B}K;            /* proves H holds K */
GOALS
   SECRET K;
END;
```