SRI INTERNATIONAL

SRI Technical Report

The CAPSL Integrated Protocol Environment

Grit Denker, Jonathan Millen and Harald Rueß
Computer Science Laboratory

SRI-CSL-2000-02, October 2000

Supported by DARPA through the Air Force Research Laboratory
Contract F30602-98-C-0258 from 14 August 1998 to 31 July 2000
Principal Investigator Jonathan Millen (650) 859-2358
DARPA Req. No. N-8-6136 Amount $972,087.00

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

The CAPSL Integrated Protocol Environment[1]

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

Abstract

CAPSL is a Common Authentication Protocol Specification Language intended to support analysis of cryptographic protocols using formal methods. CAPSL is adapted for use by various protocol analysis tools using an intermediate language, CIL. This report includes a CAPSL tutorial, the syntax of CAPSL and CIL, and the abstract rewriting model underlying CIL. Algorithms are given for translating CAPSL to CIL and for CIL rule optimization. Methods are given for integration of CAPSL and CIL with analysis tools, specifically PVS, Maude, and Athena, and for protocol analysis using PVS and Maude.

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

Contents

Chapter 1

Introduction

1.1 Cryptographic Protocols

In computer networks, cryptography is used to protect private messages and to authenticate the source and content of messages. Security objectives may take a sequence of several messages, that is, a protocol, to achieve. A protocol can be specified diagrammatically as in Figure 1.1. This particular protocol is supposed to establish a session between *principals* A and B in such a way that each principal authenticates the identity of the other principal, and they share two session-specific secrets N_a and N_b. (This is actually not the whole protocol in [NS78], but just the handshake that comes after an earlier part in which the public keys are obtained.)

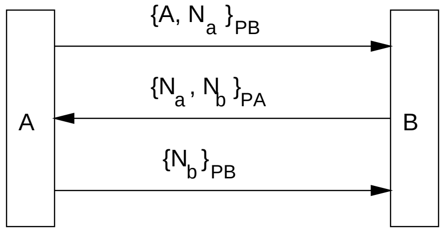

Figure 1.1: Needham-Schroeder Public-Key Protocol

The bracketed term $\{A, N_a\}_{PB}$ represents the encryption of the concatenation of A and N_a using the public key of B. It is assumed here that A has previously obtained B's public key and that only B has the corresponding secret key, and vice versa for B. The message fields N_a and N_b are *nonces,* meaning that they are *fresh*, in the sense that they have not been used before by the principal that originates them. If they are large enough and randomly generated, they could be used as keys to encrypt subsequent messages.

The secrecy claim is based on the argument that A has given N_a directly only to B, because only B could have decrypted the message in which N_a was introduced. Similarly for B and N_b. The protocol also provides entity authentication, i.e., evidence that the other principal is currently actively participating in the protocol, because it includes acknowledgments from B and A containing the nonces they received.

The same protocol is often represented in a more algebraic style, like this:

$$\begin{aligned}&A \rightarrow B : \{A, N_a\}_{PB}\\&B \rightarrow A : \{N_a, N_b\}_{PA}\\&A \rightarrow B : \{N_b\}_{PB}\end{aligned}$$

CAPSL is an outgrowth of this algebraic message-list style.

1.2 Message Modification Attacks

There is a *message modification* attack on the Needham-Schroeder protocol, found by Lowe [Low96]. Message modification attacks assume that there is an *intruder* or *attacker* in the network who can intercept messages, record them, and replace them with modified or different messages, which may appear to have come from different sources. The intruder may also act as a legitimate principal, either because he is one, or because he has somehow obtained a long-term secret key of one. Lowe's attack is illustrated in Figure 1.2.

In this figure, the center column represents the intruder playing two roles. One role is as himself, principal X, responding to A in the left-hand session of the protocol. The intruder is also *masquerading* as A in the right-hand session of the protocol, indicated with (A) in parentheses. There is a security breach in the right-hand session, because B ends up believing he has been talking to A, and that N_b is shared only with A.

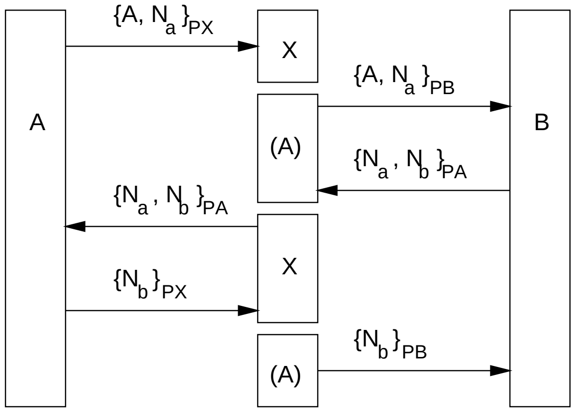

Figure 1.2: Lowe's Attack

1.3 Specification and Analysis Tools

The existence of message modification attacks led to the development of methods to detect them. Several approaches have been developed, as represented by papers such as [MCF87, Mea91] on goal-directed state search tools implemented in Prolog, [Kem89, Pau98] on the application of general-purpose specification and verification tools, [BAN90, GNY90] on specially designed logics of belief, and [Ros95, Low96, CJM98] on the application of model-checking tools. This is far from a complete list of papers on the subject.

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. This approach was taken with an earlier version of CAPSL [Mil97]; ISL, supporting an application of HOL to an extension of the GNY logic [Bra97]; Casper [Low98], for the application of FDR using a CSP model-checking approach; and Carlsen's "Standard Notation" [Car94], which was translated to per-process CKT5 specifications.

A proposal for CAPSL 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. An overview of the CAPSL and CIL environment was given in [DM00]. Current documentation, the translator, and other resources are available on the CAPSL Web site [Mil00b].

1.4 CAPSL Features

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 in auxiliary abstract data type specifications. Specifications for some popular operators, representing the abstract features of cryptosystems like DES, RSA, and Diffie-Hellman, are included in a prelude file supplied with the CAPSL translator.

Sometimes the protocol requires computations and tests that are not conveniently expressed using just the message sequence. In CAPSL, one can insert assignment statements and equations representing computations and tests.

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 message idealizations, to help support belief logic analysis.

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

1.5 The Intermediate Language CIL

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 1.3. CAPSL is parsed and translated to CIL, and there are different translators, called *connectors,* 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. CIL uses multiset term rewriting rules that permit state changes to be presented concisely, and in a way that closely matches the requirements of analysis tools. This approach was influenced by an analysis example using Maude, by Denker, Meseguer, and Talcott, presented at a LICS '98 workshop [DMT98a], and by Mitchell's multiset rewriting formulation, presented at a Computer Aided Verification workshop in 1998, and also later, in more detail, in [CDL$^+$99].

1.6 Summary of This Document

Chapter 2 introduces the CAPSL language with a tutorial including a sequence of simple examples. It then goes on to present the elements of the

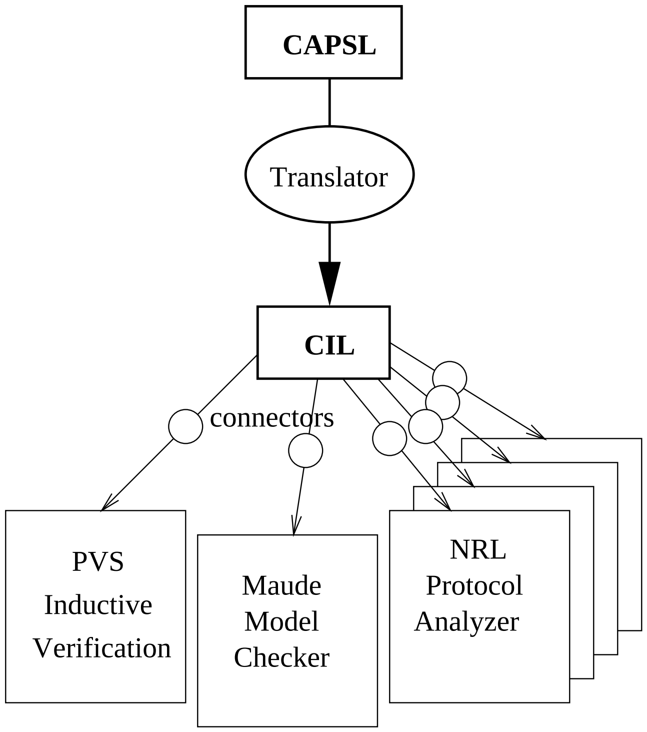

Figure 1.3: Overview of the Environment

syntax systematically. Chapter 3 describes CIL and its relation to the underlying abstract rewriting model. It also presents the algorithm for translating CAPSL to CIL, and in particular the way the rewrite rules are generated. Chapter 4 explains the optimization step, which reduces the number of rewrite rules almost in half. Then, Chapter 5 addresses the integration of CAPSL and CIL with analysis tools, using connectors. Analysis techniques for PVS and Maude are summarized, and the connector to Athena is described. There is a short conclusion, Chapter 6. The report has several appendices containing examples and reference information.

Chapter 2

CAPSL

A CAPSL specification is made up of three kinds of modules: *typespec*, *protocol*, and *environment* specifications, usually in that order. Typespecs declare cryptographic operators and other functions axiomatically. Environment specifications are optional; they are used to set up particular network scenarios for the benefit of search tools. Some standard typespecs in a prelude file are automatically utilized by the CAPSL translator, so many protocols can be specified with only a protocol module.

This introduction to the CAPSL language begins with a tutorial sequence of protocols designed to illustrate the basic features of CAPSL.

2.1 CAPSL Tutorial

Here is the simplest example of a protocol specification.

```
PROTOCOL Simple1;
VARIABLES
   A: Principal;
MESSAGES
   A -> A: A;
END;
```

Protocol Simple1 has only one message, in which principal A sends its name to itself. As in a strongly typed programming language, variables must be declared and typed. Principals are objects that can occur as the source or destination of a message.

Here is a slightly more complex example.

```
PROTOCOL Simple2;
VARIABLES
   A, B: Principal;
ASSUMPTIONS
   HOLDS A: B;
MESSAGES
   A -> B: A;
END;
```

The `HOLDS` declaration states that the process executing on behalf of A has been initialized with the principal B chosen as the responder. Read it as "A holds B." If the `HOLDS` assumption is omitted, the CAPSL translator will complain that sender of the first message does not know the receiver address. It is unnecessary to say `HOLDS A: A` because, by convention, principals always hold themselves.

2.1.1 Encryption

Certain types of principals possess long-term keys. PKUser is a subtype of Principal possessing a public key pair. If A is of type PKUser, `pk`(A) is its public key and `sk`(A) the corresponding private (secret) key. Thus, A could encrypt its message to B as follows:

```
PROTOCOL Simple3;
VARIABLES
   A, B: PKUser;
ASSUMPTIONS
   HOLDS A: B;
MESSAGES
   A -> B: {A}pk(B);
END;
```

The notation {*field*}*key* is syntactic sugar for the function call `ped`(*key*,*field*). The function `ped` is the standard abstract public key encryption and decryption function. (If the key is a symmetric key, the syntactic sugar expands internally into a call on `se`, the standard abstract symmetric key encryption function, instead.)

2.1.2 Fresh Keys

Session keys are usually assumed to be fresh, generated in some way that ensures (up to a cryptographically unlikely coincidence) that each new one has not been used before. To be useful as a key, the new value should be *unguessable.* Sequence numbers, for example, are fresh but not unguessable. Here is an example of session key generation:

```
PROTOCOL Simple4;
VARIABLES
   A, B: Principal;
   K: Skey, FRESH, CRYPTO;
ASSUMPTIONS
   HOLDS A: B;
MESSAGES
   A -> B: {A}K;
END;
```

In this example, Skey is a symmetric key type, and the declaration of K has two keywords `FRESH` and `CRYPTO` called *properties.* The `CRYPTO` property indicates unguessability.

Simple4 is not useful because B does not hold K and cannot decrypt the message to obtain A. In CAPSL, this protocol specification is semantically illegal because it implies that B decrypts the message, and the translator will complain that the message is not "receivable" by B. Declaring that B holds K does not work, because, in the first message, A cannot generate K as a fresh value if it is already held by B, and the translator complains accordingly. But there is a way to specify that B does not try to decrypt the message.

2.1.3 The Lowe %-Operator

The author of the protocol can specify that B accepts the encrypted term without attempting to decrypt it, by declaring a variable in which B stores the received value. The different views of the message – the encrypted form seen by A and the atomic form seen by B – are separated by the % operator, which was introduced by Lowe in Casper [Low98]. We can see how the % operator is used in this version of the protocol:

```
PROTOCOL Simple5;
VARIABLES
   A, B: Principal;
   K: Skey, FRESH, CRYPTO;
   F: Field;
ASSUMPTIONS
   HOLDS A: B;
MESSAGES
   A -> B: ({A}K)%F;
END;
```

The type Field is the supertype of all types that can be used as message fields, including principals, keys, and terms constructed by encryption and concatenation.

The %-operator has a weaker binding precedence than encryption, so, for example, `({A}K)%F` can safely be written as `{A}K%F`.

2.1.4 Address Convention

Suppose we wish to extend Simple5 to a longer protocol in which B replies to A with F.

```
PROTOCOL Simple6;
VARIABLES
   A, B: Principal;
   K: Skey, FRESH, CRYPTO;
   F: Field;
ASSUMPTIONS
   HOLDS A: B;
MESSAGES
   A -> B: {A}K%F;
   B -> A: F;
END;
```

The reply message is unacceptable to the CAPSL translator because "sender does not know receiver address." The problem here is that since B can't decrypt the message, B has not learned the value of A. By convention, the source address of the message is not considered part of the content, and is not readable by the receiver of the message. Realistically, this seems reasonable because, although the same "Principal" type is used in the abstraction in both the address and content portions of the message, implementations distinguish an address specification – such as an IP address – from a subject name, which is a text string chosen to be meaningful in the application context. Protocols presented in the literature are inconsistent with regard to this convention.

2.1.5 Goals

In order to analyze the security of a protocol, there must be a statement of its objectives. In CAPSL, there is a `GOALS` section to express secrecy and authentication claims. In the following simple example protocol, we might imagine that the designer intended for K to be a secret shared only by A and B, and that when B receives it, B can be assured that it was sent by A.

These two goals are stated as `SECRET` and `PRECEDES` assertions. A `SECRET` assertion says that the value of a variable generated by its nominal originator cannot be obtained by the intruder (unless the intruder is acting in one of the legitimate roles of the protocol). A `PRECEDES`$A, B|V_1, V_2, ...$ assertion says that if B reaches its final state, then A must have reached a state that agrees with B on $V_1, V_2, ...$

```
PROTOCOL Simple7;
VARIABLES
   A, B: Principal;
   K: Skey, FRESH, CRYPTO;
   F: Field;
ASSUMPTIONS
   HOLDS A: B;
MESSAGES
   A -> B: {A,K}pk(B);
GOALS
   SECRET K;
   PRECEDES A: B | K;
END;
```

In this protocol, K as generated by A is kept secret, but it might not reach B. The message received by B could have been forged by the intruder. Thus, the value of K received by B is not necessarily from A, so the PRECEDES goal would fail.

2.1.6 The Needham-Schroeder Public Key Handshake

This tutorial concludes with the CAPSL specification of the Needham-Schroeder public key handshake mentioned in the Introduction. There is a type Nonce used in this protocol which is assumed implicitly, by convention, to have the property FRESH (but not necessarily CRYPTO).

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

2.2 Types and Declarations

Typespecs define the types and operators used in protocol specifications. There is a subtype relation that places types in a hierarchy. Both typespecs and protocol specifications can declare types, constants, functions, and variables. The difference is that declarations appearing in typespecs are universal and re-usable, while those in protocol specifications are specific to a protocol. Before describing typespecs protocol specifications, we present the type hierarchy and show the kinds of declarations that may appear in either typespecs or protocol specifications.

2.2.1 The Type Hierarchy

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. Field is a subtype of the universal type Object, and there are other types of objects that are not used as message fields, such as Role, Spec and Boolean. A portion of the type hierarchy is shown in Figure 2.1.

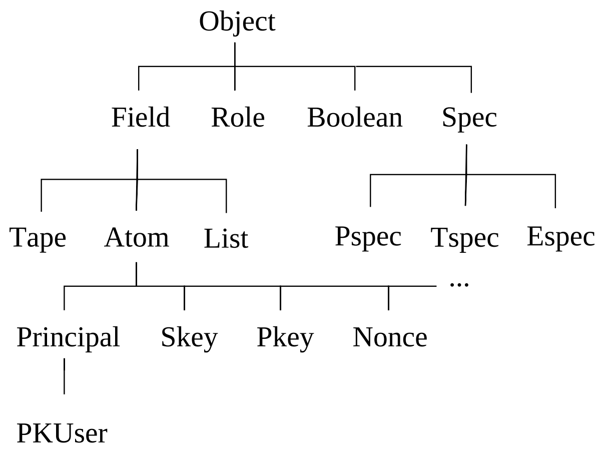

Figure 2.1: The Type Hierarchy

In principle, all functions used in CAPSL and the data types they operate on must be specified axiomatically in typespecs. The types that are shown in the hierarchy and the most commonly used encryption operators are included in the prelude. The current prelude is given in Appendix B.

2.2.2 Declarations

Declarations include IMPORTS, TYPES, VARIABLES, FUNCTIONS, and CONSTANTS, in no particular order except that identifiers must be declared before they are used.

IMPORTS. An IMPORTS declaration names one or more specifications (this is one reason why specifications are named) and indicates that the declarations contained in them or imported by them are to be used as though they were included in the present specification. An IMPORTS declaration permits the CAPSL translator to process a sequence of specifications and check that imported specifications have occurred earlier in the sequence. The CAPSL translator assumes that user specifications implicitly import the whole prelude, so that it is not necessary to import any specification in the prelude explicitly.

Importation of declarations brings all symbols into the same context. One may not, for example, declare the same function or variable twice; a "duplicate declaration" error message will result. There are three exceptions to this, as noted below, for function overloading, function refinement, and dummy variables.

TYPES. A type declaration `TYPES` $T_1, ... : T; ...$ introduces new types $T_1, ...$ and indicates that they are subtypes of T. The supertype is optional, and if it is left out, the new types are assumed to be subtypes of Atom (a generic fixed-length field type).

VARIABLES. A variable declaration `VARIABLES` $V_1, ... : T, P_1, ...; ...$ introduces protocol variables of type T, optionally with properties $P_1, ...$ The `FRESH` and `CRYPTO` properties were mentioned above; others will be introduced where they are relevant.

A variable declared in a typespec is a dummy variable, and the same variable may be redeclared as the same type in another typespec. A variable declared in a protocol specification is a protocol variable and it may not also be declared elsewhere. (A future version of the CAPSL translator may treat dummy variable declarations as local to permit redeclaration in another context.)

FUNCTIONS. A function declaration `FUNCTIONS` $F_1(T_1, ...) : T_2, P_1, ...; ...$ declares the type signature of one or more functions. Function values may have the properties `PRIVATE`, `ASSOC`, and `COMM`. Private functions are discussed below in the typespec section. The `ASSOC` and `COMM` properties designate associative and commutative binary functions, to obviate axioms for that purpose.

The same function name may be re-used for another function for the sake of either overloading or refinement. Overloading means that the function name is used with a different signature that does not overlap with an earlier declaration, such as different forms of addition for Skeys and Booleans. Refinement means that domain restriction implies a range restriction, such as the refinement of encryption from fields to atoms, shown in the next subsection.

CONSTANTS. A constant is essentially a function with no arguments. A constant declaration has the form `CONSTANTS` $C_1, ... : T_1; ...$.

2.2.3 Typespecs

A typespec consists of some declarations, followed optionally by some axioms. Typespecs usually introduce a new type and some functions defined on it, but in some cases they merely extend an existing typespec by defining new functions on existing types.

Here is an example of some related typespecs found in the prelude.

```
TYPESPEC PKEY;
TYPES Pkey;
END;

TYPESPEC SPKE;
IMPORTS PKEY;
FUNCTIONS
    ped(Pkey, Field): Field;
    ped(Pkey, Atom): Atom;
END;

TYPESPEC PPK;
IMPORTS SPKE;
TYPES PKUser: Principal;
FUNCTIONS
    pk(PKUser): Pkey;
```

```
        sk(PKUser): Pkey, PRIVATE;
    VARIABLES
        A: PKUser;
        X: Field;
    AXIOMS
        ped(sk(A),ped(pk(A),X)) = X;
        ped(pk(A),ped(sk(A),X)) = X;
        INVERT ped(pk(A),X): X | sk(A);
        INVERT ped(sk(A),X): X | pk(A);
    END;
```

The first typespec declares a data type Pkey (public key). New types are subtypes of Atom unless otherwise indicated. The second typespec defines public-key encryption using a single encryption/decryption function `ped`. This function has two type signatures, a more general one for encrypting fields, and a more specific one that says that atomic fields are encrypted to atomic fields. This is an example of function refinement.

The encryption/decryption cancellation property for public-key encryption is not stated in this typespec, because key pairs will be generated by functions associated with Principal subtypes.

Typespec PPK defines a subtype PKUser of Principal. PKUsers have two functions defined for them giving their permanent public and secret keys.

Functions are normally public or universal, in the sense that anyone can compute them, given the argument values. This is not what we want for the secret-key function, for if anyone could compute the secret key `sk`(A) just by knowing A, the secret key would hardly be secret. Hence the `PRIVATE` property.

If the first argument of a function is of type Principal, it can be declared with the `PRIVATE` property to indicate that the value is available only to the principal named in the first argument. Thus, only Alice can find `sk`(Alice).

There are four axioms in PPK. The first two say that `sk`(A) and `pk`(A) are inverse keys with respect to `ped`, in either order. The CAPSL translator does not "understand" these axioms; it simply passes them on through CIL to the analysis tools. However, the invertibility properties of `ped` are also expressed in the corresponding `INVERT` statements. These are used by the CAPSL translator to check implementability of specifications. `INVERT` $t : x|y$ means that any party (either legitimate or the intruder) holding term t containing a variable x can compute x provided that it holds y. y could be a list of terms. The use of `INVERT` statements is covered in more detail in Chapter 3.

The use of typespecs to define subtypes of Principal with functions to look up their long-term keys is an important, original stylistic aspect of CAPSL.

2.3 Protocol Specifications

A protocol specification has the form:

```
PROTOCOL name;
declarations
ASSUMPTIONS
assumptions
MESSAGES
messages and actions
GOALS
goals
END;
```

There is one special kind of declaration that occurs only in protocol specifications: `DENOTES` declarations.

2.3.1 DENOTES Declarations

`DENOTES` declarations allow a variable to be defined as the value of an expression. This is helpful in protocols where there are certificates or tickets or public values with a complex structure, and we want to define them initially and use a variable for them in the body of the protocol. A `DENOTES` declaration section can declare more than one variable. It has the form `DENOTES` $V = e : A, ...; ...$ where e is a term and A is a principal. More than one principal, or none, may be listed.

A declaration $V = e : A$ is treated as an assignment action that is executed by A when V is first used by A. This might happen when A is putting V into a message, or when A receives a message purportedly containing V so that it can make a comparison. A `DENOTES` action is used only once by each agent, since V is held thereafter.

It is possible to omit the principal from the `DENOTES` declaration, as in `DENOTES` $V = e$. In this case, all principals will use this action.

`DENOTES` equations must be placed in logical order. That is, if there is a `DENOTES` equation $V = f(X)$ and also a `DENOTES` equation for X, the equation for X must appear earlier.

The variable on the left in a `DENOTES` declaration must be declared separately. It is a real protocol variable, not merely a placeholder for macro substitution.

`DENOTES` declarations are helpful when the same value is computed in different ways by different principals. One example of this is in generating a common key via Diffie-Hellman agreement. Another example is when a long-term key must be looked up using a different private function by each principal.

2.3.2 Assumptions

Syntactically, assumptions include statements and certain special forms. Assumption statements are boolean-valued terms or equalities. The special form most commonly used as an assumption is the `HOLDS` form.

CAPSL also allows statements and other assumptions to be qualified with the belief operator, e.g., `BELIEVES` A : `BELIEVES` B : `HOLDS` $A : K$, read: "A believes that B believes that A holds K." This syntax is included to help support belief logic applications of CAPSL. Belief assumptions are useful only in conjunction with a suitable modal logic to infer belief goals.

There is also a `KNOWS` operator. This does not refer to values, as `HOLDS` does. Instead, it is Hintikka's epistemic logic operator, related to `BELIEVES`. The relationship is that `KNOWS` $A : \phi$ is equivalent to $\phi \wedge$ `BELIEVES` $A : \phi$, i.e., belief plus truth.

2.3.3 Messages

The `MESSAGES` section of a protocol specification is a sequence of messages, among which *actions* may be interleaved. The `MESSAGES` section may end with a subprotocol invocation. These are complex enough subjects so that a separate section is allocated to discuss them.

2.3.4 Goals

The GOALS section states the security objectives for the protocol. Syntactically, the same assertions that are legal as assumptions are legal as goals. However, in the GOALS section one expects to see secrecy and authentication assertions.

Secrecy. A SECRET assertion has the form SECRET $V : P_1, ...$ It says that the protocol variable V is secrets shared only by the principals $P_1, ...$ The list of principals may be omitted, in which case it is understood that the secrets are shared by all of the principals playing legitimate roles in the protocol session. The semantics of secrecy assertions is discussed in depth in [MR00]. The CAPSL translator does not yet (at this writing) introduce "spell" events as described in that paper; it merely parses the SECRET assertion and passes it on in abstract syntax.

Precedence and Agreement. A PRECEDES goal has the form PRECEDES $A, B|V_1, V_2, ...$ Intuitively, this says that if some instance of the B role reaches its final state, it agrees with some instance of the A role on A, B, V_1, and V_2.

Agreement is like precedence except that there is no existence claim. For example, the goal AGREE $A, B : V_1, ...|W_1, ...;$ says that if there is any instance of A that agrees with B on $A, B, W_1, ...$, then it must agree on $V_1, ...$ also.

2.4 The MESSAGES Section

The message format is straightforward, but there are some interesting features in the presentation of message fields. We discuss those below. Besides messages, the MESSAGES section may contain actions and subprotocol invocations.

2.4.1 Message Format

A message has the form

id. sender -> receiver: field, ...;

The sender and receiver must be protocol variables of type Principal, and the content fields are terms of type Field. The message *id* (and its associated period) are merely decorative and optional. Some infix operators and other notational conveniences have been introduced to permit CAPSL messages to look like those in the literature. The existing infix operators fall into four categories: concatenation, encryption, arithmetic, and the %-operator.

Concatenation. A sequence of fields may be concatenated into a single longer field, usually for the purpose of having them encrypted together. Curly brackets { , } and square brackets [,] denote different kinds of concatenation, which are translated into different functions, `cat` and `con` respectively. `cat` is associative and `con` is not.

Both `cat` and `con` are binary. Longer concatenations are parsed under the assumption that right association is intended. Thus, $[a, b, c]$ is parsed as $[a, [b, c]]$.

Associativity of concatenation matters when we try to decompose a concatenation. With non-associative `con`, the first component of a concatenation `[[A,B],C]` is `[A,B]`. With associative `cat`, the first component of `{{A,B},C}` would be A, unless A is itself a concatenation.

To deal with this question we differentiate between *atomic* fields, which form the subtype Atom of Field, and those fields that are expressible as a concatenation of smaller fields. The first component of a `cat` concatenation is the first atomic component. Most types – all types in the prelude except Field and List – are subtypes of Atom.

Note. A message `A -> B: {C,D}` can be received by B only if C is atomic. If C is not atomic, B cannot parse the concatenation from left to right - it won't know where C stops and D begins. The translator generates an error message if this condition is not met.

Encryption. Putting a key after a bracketed expression denotes encryption, using `ped` if the key is of type Pkey or `se` if the key is of type Skey. Thus, the expression {`A, K`}`pk(B)` is interpreted as `ped(pk(B), cat(A, K))`.

A key after brackets also indicates encryption even without a concatenation, so that {`X`}`K` is interpreted as `se(K, X)`.

Decryption with `sd` is indicated with a prime, for example, {`X`}'`K`.

The same encryption functions are invoked with square brackets; the only difference is that the `con` operator is used for the concatenation. There is no difference between {X}'K and [X]'K.

Arithmetic. CAPSL permits infix arithmetic operators +, -, *, /, and ^ with type Skey. These are automatically translated to functions `pls`, `mns`, `tms`, `div`, and `exp`, which appear in the prelude. The prelude does not attempt to axiomatize these functions. It is assumed that any verification tool that can deal with these functions has its own understanding of them, and the connector for that tool will rename them appropriately.

In protocols, arithmetic is usually finite-field arithmetic with respect to some modulus. The value of the modulus may be important for cryptanalysis but it often does not matter for protocol analysis. An example of the use of these operators, the SRP protocol, appears in Appendix C.

Arithmetic is used most often to compute symmetric keys, and that is why these operators are defined on type Skey. It may be desirable to broaden the domain of arithmetic to all atomic types, with some protection against mixing different types.

Some protocols add or subtract one from a nonce in a handshake response to protect against replay. They don't really need arithmetic. They can use any value-changing function. A protocol can have a declaration `FUNCTIONS inc(Nonce): Nonce` if it needs to increment a nonce for this purpose. No axioms are needed.

Lowe's %-Notation. Sometimes it is necessary to distinguish between the sender's view of a message and the receiver's view, 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 a comma separating fields.

2.4.2 Actions

An action is an statement that occurs in a message list. An equational action like $X = e$ may be either an assignment that sets X or a comparison test, depending on whether or not the agent already holds X in that state. If X is held, the equation can only be a comparison test, because variables do not change value. But if X is not held, the equation must be an assignment, since it would be undefined as a test. If there is a term other than a variable on the left, the action must be a comparison test. (However, in the future CAPSL may support assignment into an element of an array, and then the expression on the left will be an indexed variable.)

CAPSL can handle an equation with a concatenation of variables on the left, such as $\{A, B\} = e$. This is expanded into two equations, in this case $A = \texttt{first}(e); B = \texttt{rest}(e)$.

The result of a failed comparison test, incidentally, or the evaluation of any other kind of statement to `false`, is that the acting agent does not make any further state transitions. That is, it crashes silently. If the failure of the test is supposed to generate an error reply, that must be indicated explicitly in the protocol specification, using the conditional selection syntax discussed below.

Actions may also be of the form `ASSUME` *statement* or `PROVE` *statement*. These are essentially assumptions and goals, respectively, except that they are associated with intermediate states rather than with initial and final states.

2.4.3 Phrases

When processing an action, the CAPSL translator must determine which agent (actually, which role) is taking the action, so that it can determine which variables are held. The acting agent is usually apparent from the position of the action in the message list. For example, in the message-list fragment

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

it is obviously B who executes the action $X = Y$. But, what if two messages in a row are sent by the same agent? For example,

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

Now it is not clear whether the action is taken by B after receiving the first message, or by A before sending the next. The ambiguity is resolved in CAPSL by inserting a slash "/" as a phrase divider to separate receiver actions from sender actions. In this case, if we intended that the action be performed by B, we would write:

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

The term "phrase" refers to a message and the actions before and after it by its sender and receiver. Invocations and selections, explained in the next subsection, are also phrases.

2.4.4 Subprotocols and Selection

Some complex protocols are actually frameworks that guide a communication session through a sequence of "exchanges" or subprotocols. A subprotocol is a way to encapsulate a logically cohesive sequence of messages. In CAPSL, a subprotocol is just a separate protocol module.

A protocol P_1 may invoke another one, say P_2, by using an `INCLUDE` P_2 phrase (called an *invocation*) in place of a message. The two protocol specifications would occur in the order P_1, P_2, so that variables defined in P_1 may be imported and used in P_2. The name P_2 should be declared in P_1 as a constant of type Pspec. The outline of invocation use is something like this:

```
PROTOCOL P1;
CONSTANTS P2: Pspec;
...
MESSAGES
```

```
  ...
  INCLUDE P2;
END;

PROTOCOL P2;
IMPORTS P1;
...
END;
```

Conditional Selection. The selection of later subprotocols may be conditional on data values or agreements reached in earlier message exchanges. Another reason for conditional branching in a protocol might be to provide an error reply as an alternative to a normal continuation after some test fails. In CAPSL, a conditional expression selecting alternative invocations is called a *selection.* An alternative in a selection could be any kind of phrase, so it could be a message, an invocation, or a nested selection. The SSL example in Appendix C illustrates how this is done. Here is an outline of how a selection is used:

```
PROTOCOL P1;
CONSTANTS P2: Pspec;
...
MESSAGES
  ...
  IF A = B THEN INCLUDE P2;
  ELSE INCLUDE P3; ENDIF;
END;

PROTOCOL P2;
IMPORTS P1;
...
END;

PROTOCOL P3;
IMPORTS P1;
...
END;
```

There is a problem, in principle, with continuing a protocol with more messages after a selection with alternative subprotocols. Different subprotocols may cause different sets of program variables to become held. The legality and meaning of subsequent messages is then ambiguous.

Presently, CAPSL deals with this problem in a rather draconian way by requiring that no statement may follow an `INCLUDE`, even when there is no selection. Invocations can be chained, however, to achieve sequencing. For example, protocol P_2 in the outlines above could contain another invocation at the end of its `MESSAGES` section.

A more advanced treatment of subprotocol invocation would be to give them arguments so that they could have their own protocol variable contexts, just as program subroutines do. At the moment, this feature would probably outstrip the capabilities of protocol analysis tools. Analysis is simpler if we assume that subprotocols do not "return," they just take over from the parent session.

2.5 Environment Specifications

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. CAPSL specifications can include `ENVIRONMENT` specifications containing this kind of information. Each environment specification sets up a different scenario for analysis. An environment specification contains declarations, one or more `AGENT` sections, and, optionally, any of an `EXPOSED` section, an `AXIOMS` section, and an `ORDER` section.

An environment specification defines constants for principals and perhaps other values like compromised keys. The specification constructs agents by naming the principal, role, and initial values for each agent.

Declarations to name principals and other constants could be placed in this section. For example, suppose we want two principals, Alice and Bob, taking the usual 'A' and 'B' roles, and Mallory as a dishonest principal. We might set that up like this:

```
ENVIRONMENT Test1;
  IMPORTS NSPK;
  CONSTANTS
    Alice, Bob: PKUser;
    Mallory: PKUser, EXPOSED;
  AGENT A1 HOLDS
    A = Alice;
    B = Bob;
  AGENT B1 HOLDS
    B = Bob;
  EXPOSED
    {Bob}sk(Alice);
END;
```

An environment specification imports the protocol specification it applies to, in order to refer to its protocol variables (A, etc.). Agents are named (this is an implicit declaration of a constant of type Agent) and constants must be given as values for the protocol variables initially held by the agent, as required in the protocol assumptions. The first equation, by convention, names the principal that owns the agent, so that the role of the agent can be determined. Nonces could be assigned values here or not, depending on the needs of the analysis tool.

When several environment specifications are included to analyze different scenarios, each one can import previous specifications to take advantage of the constant declarations in them. Agent declarations are not 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.

A principal with an `EXPOSED` property is one whose private data is all held initially by the attacker. In this example, if Mallory is `EXPOSED`, the values of private functions with Mallory as the first argument (such as, for example, `sk`(Mallory)) would not have to be added to the `EXPOSED` list, because they are implicitly assumed to be exposed.

Agents are, by default, assumed to run concurrently. CAPSL permits an `ORDER` section to specify some series-parallel sequencing of agents, for the benefit of search tools that could save time when such a restriction is assumed. For example, we might say: `ORDER (A1; A2)||B1` to mean that agent A_2 does not start until A_1 ends, but that sequence runs concurrently with B_1.

An environment specification may have an `AXIOMS` section for assumptions about its constants, e.g., `AXIOMS sk(Alice) = SKa`.

Chapter 3

CIL

3.1 Multiset Rewrite Rules

Support for multiple analysis tools is accomplished through the CAPSL Intermediate Language (CIL) [DM99b]. The purpose of CIL is to unambiguously define the meaning of a protocol specification. CIL also acts as an interface through which protocols specified in CAPSL can be analyzed using a variety of tools.

The challenge for the design of CIL was to make it general enough and expressive enough to represent a wide range of protocols, and yet at a low enough level to be close to the representation used by most verification or model-checking tools. Many such tools share a specification style that incorporates state-transition rules specified in a pattern-matching style, with symbolic terms to represent encryption and other computations. There is usually a separate and fairly standard intruder model.

As an example of the use of pattern matching, 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)`. This implies that A must decrypt the message content and confirm that the result is a concatenation of two fields of type Nonce. Furthermore, if A already holds a value for N_a or N_b, it will compare that with the one in the message. "Accepting" a message means that A will undergo a state transition as a result of receiving it.

The commonality in the abstract symbolic treatment of protocols was recognized and codified in the Cervesato, *et al* meta-notation, in which state transitions are expressed with multiset rewriting (MSR) rules [CDL+99]. The MSR notation was adapted for CIL. Their notation, according to the authors, could be regarded as either an extension of multiset rewriting with a kind of existential quantification, or a Horn fragment of linear logic. The simplicity and generality of this formalism made it suitable to serve as the language in which to express the semantics of CAPSL. Furthermore, the term-rewriting aspect corresponded well with the analysis approach taken by Denker, Meseguer, and Talcott with Maude [DMT98b]. CIL may be regarded as a notational variant of the MSR formalism in which certain specific conventions have been used to set up protocol models derived from CAPSL specifications.

3.1.1 The MSR Protocol Model

The MSR formalism uses transition rules of the form

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

where each F_i and G_j is a "fact." Facts are atomic formulas of the form $P(t_1, ..., t_r)$ where P is a predicate symbol and the arguments t_i are terms. A term is constructed from typed constants, variables, and function symbols. Free variables are implicitly universally quantified.

The state of a system can be represented by a multiset of facts. 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 in linear logic has a special meaning. Quantified variables are instantiated with fresh (unused) constants. This behavior is used to model generation of nonces.

In protocol modeling, facts are used to express the entrance of a process into a state, or the transmission of a message. In MSR, a state is represented by a fact $A_i(...)$ where A is the name of a protocol variable of type Principal, i is a state label, usually an integer, and the arguments are the "memory" of the agent in that role and state. A message (in our dialect) is a fact

$M(a, b, t)$ where a and b are principals and t is a term representing the message content. Another kind of fact can represent attacker knowledge.

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 rule

$$\longrightarrow A_0(A, B), B_0(B)$$

creates two facts representing the initial state of two new agents. Since A and B are variables of type Principal, this rule can initiate sessions between any pair of principals. Thus, $A_0(A, B)$ says that an agent playing the 'A' role of the protocol is in a state labeled 0 and is ready to begin a session between principal A, which owns the agent, and principal B.

The message `A -> B: A, {N}SK(A)` would result in at least two transitions, one for the sender A and one for the receiver B. The A transition would be:

$$A_0(A, B) \longrightarrow (\exists N) A_1(A, B, N), M(A, B, \{A, \{N\}\mathtt{sk}(A)\}).$$

The B transition would be:

$$B_0(B), M(X, B, \{A, \{N\}\mathtt{sk}(A)\}) \longrightarrow B_1(B, A, N).$$

The X in the sender position of the received message is a new variable. We assume here (like Paulson [Pau98]) that message facts indicate the true sender of the message, but that receiver transitions can depend only on the content of the message, and therefore the sender field is not matched with any other variable.

3.1.2 CIL Rule Syntax

MSR rules appearing in the output of the CAPSL translator are expressed in CIL syntax, in a uniform functional notation. All state facts are of the form `state`(*role, num,* `terms`(...)), where *role* is a role constant constructed from a principal variable name, such as `roleA`, and *num* is a state label, usually a natural number. The memory items are arguments of the `terms` list. Encryption and concatenation are expressed using the functional forms declared in the prelude or other typespecs. Messages are `msg` facts.

So, for example, the transition

$$A_0(A, B) \longrightarrow (\exists K) A_1(A, B, K), M(A, B, \{A\}K)$$

would appear in CIL as

```
   rule(facts(state(roleA,0,terms(A,B))),
        ids(K),
        facts(state(roleA,1,terms(A,B,K)),
              msg(A,B,terms(se(K,A)))))
```

The syntax of CIL, which includes other items besides rules, is given in Appendix A.2.

3.2 Translator Overview

3.2.1 CIL Output

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 unraveling a message-list protocol description into a set of rewrite rules. Besides the rules, the output of the translator includes symbol table information and other information that will be used by connectors and analysis tools.

The output of the CIL translation has several parts:

1. slot table
2. symbol table
3. axioms
4. localized assumptions
5. protocol rewrite rules
6. localized goals
7. environment information

The actual output of the CAPSL translator is a text file expressing this information in the abstract syntax of CIL, using a functional notation. A CIL specification has the form:

```
CILspec(
  symbols(symbol(...),...),
  slots(slot(A,roleA,1),...),
  axioms(...),
  assums(...),
  rules(
    rule(facts(...),ids(...),facts(...)),
    ...
  ),
  goals(...),
  envs(...),
)
```

The CIL specification of NSPK can be found in Appendix D.

A symbol table entry has the form

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

where *ident* is the symbol name, *status* is the kind of symbol, one of `op` for a function or constant, `pvar` for a protocol variable, `var` for a dummy variable, or `type` for a type name. The argument types are in a list of the form `ids`(...) and the properties are in a list of the form `props`(...). The symbol table contains all identifiers declared in all of the specification modules.

The slot table maps each protocol variable in the original specification to an argument position in the state predicate of each role. This is necessary for interpreting goals, agent initialization, and other statements that refer to protocol variables.

For example, if we assume that B has the value `Bob` in the initial state of an agent in role 'A', namely `state(roleA,0,terms(Alice,Bob))`, we need to know that B is the second argument in the terms list of the `roleA` state fact. This is expressed by the slot table entry `slot(B,roleA,2)`. The slot number for a program variable does not change once it is created; this convention is enforced by the way the translator generates state facts.

Axioms from typespecs and environment specifications are consolidated into a single list.

The difference between axioms and assumptions is that axioms are universal and only refer to dummy variables, while assumptions, like goals, refer to program variables and are *localized* to particular states. Thus, axioms are simply passed on as the abstract-syntax form of axioms that occur in the CAPSL specification. Assumptions and goals are expressed in the form `loc(nodes(node(`*role, state*`), ... ), `*assertion*`)`.

A CAPSL assumption is localized to the initial state, and a typical node would be `node(roleA,0)`. Declarations of such role constants are added automatically to the symbol table. CAPSL goals are localized to the final state, as determined by the translator. The *assertion* is the abstract-syntax form of the CAPSL assertion.

The CIL format for rules was summarized above, and the process for generating them is discussed in detail below in this chapter.

An environment entry has the form

`environment(`*ident*`, agents(...), exposed(...), order(...))`

where the `exposed` and `order` components may be empty, and an agent is specified as `agent(`*ident*`, eqns(eqn(`*pvar, term*`), ...))`. The identifier is just a reference constant, and the equations assign values to protocol variables. The first protocol variable listed is the principal whose variable name defines the role being played by the agent. Other protocol variables are set as required to provide initial values. The values are usually given as constants declared in the environment. The CIL symbol table includes those constants.

3.2.2 Translation Stages

The major stages in translation are the following:

1. Parsing and type checking
2. Syntax transformations
3. Rule generation
4. Local Assertions
5. Optimization

Parsing checks CAPSL syntax and produces a parse tree. Type checking confirms the consistency of type and signature declarations with each other and with terms occurring in axioms, messages and elsewhere. In the process, it replaces generic encryption expressed with bracketed terms by a choice of `ped` or `se` by checking the type of the key. It also generates a symbol table.

There are several syntax transformations:

1. `INCLUDE` phrases are expanded by replacing them by the message list of the named protocol.
2. Infix arithmetic operators are converted to functional form.
3. `cat` and `con` applications are made binary by assuming right associativity.
4. Uses of % are checked and lifted to the top level of each term. The function symbol for % is `lowe`.
5. Role constants are created for participating principals.
6. `DENOTES` equations are inserted where necessary. This is covered in more detail in the next subsection.

3.2.3 DENOTES Processing

The idea behind `DENOTES` processing is to insert an equational action into the message list when a variable with a `DENOTES` equation is seen for the first time by each principal. These modifications are made in abstract syntax to the parse tree, rather than to the original CAPSL text.

Suppose that the specification contains `DENOTES` $X = f(Y) : A$, and that the first reference to X by A is in an action, say $Z = g(X)$. This is the simplest case, and the equation for X is placed just before the action.

Even in this simplest case, we must consider that Y might have a `DENOTES` definition, and its use will recursively require the insertion of its equation, and so on. This concern is handled by (1) requiring that `DENOTES` equations be placed in logical order, so that, in this case, the equation for Y comes before the equation for X; and (2) processing the `DENOTES` equations in reverse order, so that the equation for Y will be inserted before the previously inserted equation for X.

Suppose the first reference to X by A is in a transmitted message, say `A -> B: Y, X`. As in the case of of an action, we place the equation for X just before the message.

Suppose the first reference to X by A is in a received message, say `B -> A: Y, X`. Then we cannot place the equation before the message, because it would be executed by B rather than A.

What we do, instead, is (1) replace X in the message by the right side of the equation, $f(Y)$, and then (2) insert the equation for X after the message. This is equivalent to having written

```
B -> A: Y, f(Y);
X = f(Y);
```

3.3 Abstract Rule Generation

The core of translating CAPSL to CIL is the creation of rewrite rules from messages and actions. To create rules successfully for a message, the message must be *implementable.* Two issues for implementability are *invertibility* and *computablitity* of message fields.

Each message gives rise to at least two transitions, one for the sender and one for the receiver. With respect to the sender, the translator must check whether the sender is capable of computing all the message fields–that is, whether the message is computable. For a message to be computable, the sender must hold the variables mentioned in it and be able to access any private functions used.

With respect to the receiver, the translator has to test the message for "receivability." A variable is receivable, whether it is already held or not. If it is held, the receiver performs a comparison with the prior value. If it is not held, it is learned, and a slot in the state is created for it. In the case of a functional term, receivability means that the term is either computable, so that the receiver can compare the received value to its own locally stored or recomputed value, or it is invertible, so that the receiver can decompose it, and then test or store or further decompose each extracted subterm.

The algorithm for generating rules, with definitions for computability and invertibility, is given in this section. For purposes of presenting the algorithm, we regard the translator as a finite-state machine. Its state is a set of role states, its inputs are the messages in the message list, presented in order, and its outputs are the generated rules.

3.3.1 Translator State

The state of a role is represented by a term $S(p, n, \mathbf{x})$ where p is a protocol variable of type Principal, n is a state number, and $\mathbf{x}$ is a sequence of terms held by p. Most of the terms in $\mathbf{x}$ are protocol variables, but compound terms may be present as well.

Before any message is processed, the initialization rules are generated, one for each role in the protocol. The initial role state for p has $n = 0$ and a sequence $\mathbf{x}$ that begins with p and also includes any variables declared as held by that principal in the `ASSUMPTIONS` declaration.

For our purposes in describing the translation, we represent a message as a term $M(p, q, t\%r)$ where p and q are the variables representing the sender and receiver of the message, and $t\%r$ shows the sender's version t of the message content and the receiver's version r. In CAPSL, a message can have several fields, but for simplicity we assume here that t and r are single terms. If the message has more than one field, its content could be represented as a concatenation of these fields.

3.3.2 Computability and Receivability

We begin with some necessary terminology. In general, a boldface symbol is a sequence, so $\mathbf{x} = x_1, ..., x_n$ for some n. In some contexts we will also use $\mathbf{x}$ to refer to the set of its components. $R(p)$ denotes the symbol of type Role which corresponds to a symbol p of type Principal.

Accessibility. A function $f(\mathbf{y})$ is *p-accessible* if f is not private (does not have the PRIVATE property) or f is private and $y_1 = p$.

Computability. In defining computability of a term, we assume that some terms are held–this is the set G–and we derive the set of additional variables X that are needed to compute the term. The principal p is mentioned only because of the need to test accessibility.

t is *p-computable given G with X* if

1. $t \in G$ and $X = \emptyset$ or

2. t is a protocol variable and $t \notin G$ and $X = \{t\}$ or

3. $t = f(\mathbf{y})$, $f(\mathbf{y})$ is p-accessible, each y_i is p-computable given G with X_i, and $X = \bigcup_i X_i$.

We say that t is p-computable given G if t is p-computable given G with $\emptyset$. If Z is a set of terms, we say that Z is p-computable given G with $\bigcup_{t \in Z} A_t$ if each $t \in Z$ is p-computable given G with A_t.

As an example, consider $t := \texttt{ped}(\texttt{SK}(A), N)$. t is A-computable given $\{A\}$ with $\{N\}$ because `ped` is not private, and, although `SK` is private, $\texttt{SK}(A)$ is A-accessible.

Invertibility. To define p-invertibility, we assume that there are axioms of the form $\texttt{inv}(f(\mathbf{y}), y_i, Z)$ for some operators f, where $\mathbf{y}$ is a sequence of different variables and Z is a list of terms not including y_i. An invertibility axiom states that $f(\mathbf{y})$ can be inverted to compute a value for y_i provided that the values of all terms in Z are computable. For example, $\{X\}\texttt{pk}(A)$ can be inverted to compute the value for X given $\texttt{sk}(A)$. The CAPSL concrete syntax for an invertibility axiom uses the keyword `INVERT`, and in the CIL syntax this becomes an `invertible` statement. Encyrption functions are generally invertible; look at the prelude for examples of invertibility axioms for them.

t is *p-invertible at i given G* if $t = f(\mathbf{y})$ and $\texttt{invertible}(f(\mathbf{y}), y_i, Z)$ and Z is p-computable given G.

Receivability. If a term t is a variable or constant (a function with no arguments), receiving it means to compare it with the terms in the held set G and add it to G if it is not there. If t is compound, it must be either computable or invertible, and in the latter case the components extracted from it are received recursively. This process enlarges G to H.

t is *p-receivable given G to H* if

1. t is p-computable given G and $H = G$, or

2. t is p-computable given G with $\{t\}$ and $H = G \cup \{t\}$, or

3. We have:

    (a) $t = f(\mathbf{y})$ is p-invertible at some j given G and

(b) $\mathbf{y}'$ is sequentially p-receivable given G to H', where $\mathbf{y}'$ is the maximum subsequence $y_{i_1}, ..., y_{i_k}$ such that t is p-invertible at i_j given G, and

(c) if t is p-computable given H' then $H = H'$ else $H = H' \cup \{t\}$.

Sequential receivability expresses the notion that variables learned while receiving a message can be used to compute terms received later in the same message.

$\mathbf{y} = y_1, ..., y_n$ is *sequentially p-receivable given G to H* if, for $j = 1, ..., n$, y_j is p-receivable given G_j to H_j, where $G_j = H_{j-1}$ and $H_0 = G$ and $H_n = H$.

The success or failure of the sequential receivability test depends on the order of the sequence of terms, since the held set G is augmented as part of the process. A more forgiving definition would be able to rearrange the order to find one that works, and it could be implemented by making several passes over the sequence.

As an example, consider $t := \texttt{ped}(\texttt{sk}(A), N)$. t is B-receivable given $\{A\}$ to $\{A, N, t\}$. Upon receiving t the agent in role B not only learns the nonce N but also the whole term t since t is not B-computable given $\{A, N\}$.

3.3.3 Message Rules

A message $M(p, q, t\%r)$ gives rise to two protocol rewrite rules, one for p to send the message t, and one for q to receive r. Each protocol rewrite rule is generated by a translator state transition. A transition associated with sending the message affects only the sender-role state, and the one associated with receiving the message affects only the receiver-role state.

A schema is a way of presenting a set of translator transitions in a parameterized form, independent of the particular state number and term sequence. There is a Send schema for the sender-role transition and a Receive schema for the receiver-role transition. A schema may specify conditions on the state transition; if they are not satisfied, the transition fails, and so does the translation. A schema ends with a protocol rewrite rule.

The Send schema says that if the message is computable, possibly with a set of new variables, the sender can transmit the message. The sender must also hold the identity of the receiver.

Notation. If A is a set of variables, $\mathbf{A}$ consists of the elements of A written as a sequence, in some arbitrary but consistently chosen order.

In the schemas below, a variable t is called *new* in the current translator state if t is a protocol variable, t is of type Nonce or has the `FRESH` property, and no other principal q has t in its current state. A set of new variables is also called new.

> **Send schema**:
> *Current state*: $S(p, n, \mathbf{x})$
> *Message*: $\mathrm{M}(p, q, t\%r)$
> *Condition*: $q \in \mathbf{x}$ and t is p-computable given $\mathbf{x}$ with A and A is new
> *Next state*: $S(p, n+1, \mathbf{xA})$
> *Rule*: $S(R(p), n, \mathbf{x}) \longrightarrow (\exists \mathbf{A}), S(R(p), n+1, \mathbf{xA}), M(p, q, t)$

The Receive schema says that if the message content is receivable with learned terms A, the receiver accepts the message and adds the terms in A to its state.

> **Receive schema**:
> *Current state*: $S(q, n, \mathbf{x})$
> *Message*: $M(p, q, t\%r)$
> *Condition*: r is q-receivable given $\mathbf{x}$ to H and $A = H - \mathbf{x}$
> *Next state*: $S(q, n+1, \mathbf{xA})$
> *Rule*: $S(R(q), n, \mathbf{x}), M(U, q, r) \longrightarrow S(R(q), n+1, \mathbf{xA})$

The receiver of a message cannot see the sender's address. Thus, we assume an arbitrary sender variable U of type Principal.

Example. Given the translator state $S(B, 2, [B, A])$ and the message $M(A, B, \{N\}\mathtt{sk}(A))$, the new translator state is $S(B, 3, [B, A, N, \{N\}\mathtt{sk}(A)])$ and the following CIL rule is generated to receive the message:

```
rule(facts(state(roleB,2,terms(B,A)),
           msg(Z,B,terms(ped(sk(A),N)))),
     ids(),
     facts(state(roleB,3,terms(B,A,N,ped(sk(A),N))))).
```

3.3.4 Equational Actions

The right side of an equational action is always tested for computability. Depending on the computability of the left side, the action is understood to be an assignment or a test for equality. If the left side terms are `con` or `cat`, we handle them in a particular way.

The Test schema says that if both sides of the action are computable, then the receiver performs a test. Two rules are created for this purpose. In the first transition the equation is added to the list of terms. If the test is evaluated to true, then the second transition advances the state number and deletes the equation.

Action schema (test):
Current state: $S(p, n, \mathbf{x})$
Action: $t = t'$
Condition: t and t' are p-computable given $\mathbf{x}$
Next state: $S(p, n+2, \mathbf{x})$
Rules: $S(R(p), n, \mathbf{x}) \longrightarrow S(R(p), n+1, \mathbf{x}(t = t'))$
$S(R(p), n+1, \mathit{true}) \longrightarrow S(R(p), n+2, \mathbf{x})$

The Assignment schema requires that the right side is computable and that the left side is a protocol variable that is not held by the agent. Then the agent can perform an assignment transition.

Action schema (assignment):
Current state: $S(p, n, \mathbf{x})$
Action: $y = t'$
Condition: t' is p-computable given $\mathbf{x}$ and y is a protocol variable, $y \notin \mathbf{x}$
Next state: $S(p, n+1, \mathbf{x}y)$
Rule: $S(R(p), n, \mathbf{x}) \longrightarrow S(R(p), n+1, \mathbf{x}t')$

If the left hand of the action is a variable that is not held by the acting agent, then the action is an assignment. The newly assigned term is added to the termlist and there is an associated slot table entry that relates the term to the variable y. Consequently, the next rule, if any, refers to the term as variable y.

There are two special action schemas in case the outmost function on the left side is one of the two concatenation functions. If the left side of the equality is a term using `cat` or `con`, then the action is split into two equalities, one for each component of the concatenation.

Action schema (con):
Current state: $S(p, n, \mathbf{x})$
Action: $con(y, z) = t'$
Condition: t' is p-computable given $\mathbf{x}$
Rules: $<$ rules for $y = head(t')$ $>$
$<$ rules for $z = tail(t')$ $>$

Action schema (cat):
Current state: $S(p, n, \mathbf{x})$
Action: $cat(y, z) = t'$
Condition: t' is p-computable given $\mathbf{x}$ and y is atomic
Rules: $<$ rules for $y = first(t')$ $>$
$<$ rules for $z = rest(t')$ $>$

The schema for `cat` is more restrictive since the `first` operator on `cat` is only defined if the first argument is an atom.

3.3.5 Subprotocols

The schema for selection says that the agent first has to evaluate the condition. If the condition is true, it transitions into a new state that is the starting state for all rules generated for the subprotocol P_1. If there are k transitions for p in subprotocol P_1, then the p starts from state $n + k + 3$ in the branch in which the condition did not hold true.

Selection schema:
Current state: $S(p, n, \mathbf{x})$
Phrase: if t then P_1 else P_2

$$
\begin{aligned}
\textit{Rules}\colon\ & S(R(p), n, \mathbf{x}) \longrightarrow S(R(p), n+1, \mathbf{x}(t = \textit{true})) \\
& S(R(p), n+1, \mathbf{x}\textit{true}) \longrightarrow S(R(p), n+2, \mathbf{x}) \\
& S(R(p), n+1, \mathbf{x}\textit{false}) \longrightarrow S(R(p), n+3+k, \mathbf{x}) \\
& < \text{rules from } P_1;\ p \text{ starts from } n+2;\ p \text{ has } k \text{ transitions} > \\
& < \text{rules from } P_2;\ p \text{ starts from } n+3+k >
\end{aligned}
$$

States of other agents in the protocol may be changed in the invoked subprotocols. Thus, the next states of other agents have to be also reflected accordingly in the branches of the selection.

3.4 Local Assertions

When initial conditions, 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:

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

3.5 An Attacker Model

The CAPSL translator does not generate attacker rules, because most attacker rules would be standard and built into any analysis tool that needs them. A standard attacker model would include an attacker memory fact such as $N(u)$, meaning that the attacker holds the term u. Because the attacker can intercept any message, there could be a rule $M(A, B, T) \longrightarrow N(T)$ and a similar rule for forging messages, $N(T) \longrightarrow N(T), M(A, B, T)$. There would be rules for decomposing and synthesizing messages using the available concatenation and encryption functions. A general attacker model of this kind is described in [CDL$^+$99].

The attacker should be able to compute the value of any function declared in a typespec, given its arguments, except private functions. If there is a standard ("Spy") or declared dishonest principal, the attacker can compute that principal's private values, e.g., `sk(Spy)`. The attacker can compute constants, since they are simply functions with no arguments.

Connectors should generate certain protocol-specific or scenario-specific rules for initializing the attacker, using information from the environment specification.

Initially the attacker holds all exposed terms as declared in the environment section. For instance, in the environment used as an example in Section 2.5, the exposed term {`Bob`}`sk(Alice)` results in a fact-generating rule for the attacker, in CIL syntax:

```
    rule(facts(),ids(),facts(net(ped(sk(Alice),Bob)))),
```

where `net`(...) is the CIL version $N(...)$ above.

If a principal is exposed, then all private functions defined for this principal are also exposed. If Mallory is a PKUser, there would be a rule

```
    rule(facts(),ids(),facts(net(sk(Mallory)))),
```

for example.

Since all constants are computable by the attacker, there would be rules like

```
    rule(facts(),ids(),facts(net(Alice)))
```

for all principal constants named in the environment.

For purposes of inductive proof, it is simpler to assume that all principals are held by the attacker, with a rule

```
    rule(facts(),ids(),facts(net(A)))
```

where A is a variable of type Principal. On the other hand, inductive proofs might model the attacker in a way that is equivalent to a rule model but expressed quite differently, using Paulson's `Analz` and other set closure functions [Pau98, MR00].

An environment might declare constants that are supposed to be secret, such as nonces, session keys, and perhaps symbols defined to name long-term secret keys using axioms. These constants can be declared with the `CRYPTO` property to prevent them from being given to the attacker initially.

A protocol might include variables representing nonces that are not secret, such as sequence numbers, or weak passwords. If these values are not protected in messages, the attacker will obtain them by eavesdropping, but if they are protected by encryption, there will need to be further attacker rules stating that they can be produced by the attacker, to represent guessing or routine computations.

Chapter 4

Optimization of CIL Rewrite Rules

4.1 Motivation

The basic, natural translation from CAPSL to CIL, as described Chapter 3 and [DM99a], generates two rewrite rules per message, one for the message sender and one for the message receiver. Often, however, the transition that receives a message and the one from the same agent that sends a reply can be collapsed into a single transition that does both, and MSR protocol encodings produced by hand usually have this characteristic. Successive computations by the same agent to update or enlarge its state memory can also be combined.

The optimization algorithm described in this chapter automatically implements the kind of rule combinations that would typically be done by hand. Relative to the simple message-by-message translation, this reduces the number of rules, as well as the number of states per role, by about 50%. We show that this reduction is sound in the sense that it is attack-preserving, by essentially the same definition used in [SS98]. The optimizing transformation has been implemented as a post-processing step in the CAPSL translator.

The number of rules has direct impact on the performance of state evaluation tools such as model checkers. In the model-checking approach, a finite instantiation of the protocol is tested for security breaches. For this purpose, an exhaustive search strategy enumerates all reachable states for a given initial state and tests whether they invalidate a given security property. Even for small protocols and very restricted numbers of sessions the number of states explodes. This is due partly to the fact that the intruder behavior is highly non-deterministic, and partly due to the fact that new sessions involving legitimate principals may be created and execute asynchronously. Thus, a linear reduction in the number of rules can reduce the number of states to be explored by an exponential factor.

Because optimizations are performed as a series of successive rule-combination steps, there is a question as to whether the order of combination steps affects the size of the final set of rules. We show that the optimization, considered as a reduction system, is terminating and confluent, and hence canonical, so that the final set of rules is unique.

4.2 Optimization Examples

We illustrate the optimization steps with the help of the NSPK protocol given in Section 2.1.6. The following two rewrite rules represent B's receipt of the first message and B's sending of the second message of NSPK.

$$\begin{aligned}
&rule1: \ B_0(B)\ M(X, B, \{N_a, A\}_{pk(B)}) \rightarrow B_1(B, N_a, A) \\
&rule2: \ B_1(B, N_a, A) \rightarrow \\
&\qquad (\exists N_b)\ B_2(B, N_a, A, N_b)\ M(B, A, \{N_a, N_b\}_{pk(A)})
\end{aligned}$$

Under the assumption that agents have a deterministic behavior, i.e., at most one rule is applicable in each agent state, we know that after the receiving the message from A, the only thing B can do is to reply with the second message to A. The following optimized rule combines B's behavior into a one-step transition in which B receives A's message and immediately replies to it:

$$\begin{aligned}
&rule1,2: \ B_0(B)\ M(X, B, \{N_a, A\}_{pk(B)}) \rightarrow \\
&\qquad (\exists N_b)\ B_2(B, N_a, A, N_b)\ M(B, A, \{N_a, N_b\}_{pk(A)})
\end{aligned}$$

When the two rules are combined, the original pair of rules is deleted. Optimization occurs only when there is no other way to enter state B_1, so in effect state B_1 is also eliminated.

Combining the rules in this example is straightforward since the right-hand side of the first rule and the left-hand side of the second rule are identical. More generally, for two rules R and R' to be optimizable it is necessary (though not sufficient) that the state fact on the right-hand side of R is an instantiation of the state fact on the left-hand side of R'. The next example illustrates this.

For the sake of this example, replace the message `B -> A: {Na,Nb}pk(A)` message in NSPK by a sequence of two actions, an assignment and a message transmission, so that the message list is:

```
MESSAGES
  A -> B: {A,Na}pk(B);
   T = {Na, Nb};
  B -> A: {T%{Na,Nb}}pk(A);
  A -> B: {Nb}pk(B);
```

This message list yields the following CIL rules for B transitions.

$$
\begin{aligned}
&rule1: \ B_0(B)\ M(X, B, \{N_a, A\}_{pk(B)}) \to B_1(B, N_a, A)\\
&rule2: \ B_1(B, N_a, A) \to\\
&\qquad (\exists N_b)\ B_2(B, N_a, A, N_b, \{N_a, N_b\})\\
&rule3: \ B_2(B, N_a, A, N_b, T) \to\\
&\qquad B_3(B, N_a, A, N_b, T)\ M(B, A, \{T\}_{pk(A)})
\end{aligned}
$$

In this case, besides combining $rule1$ and $rule2$, we can also combine $rule2$ and $rule3$, since $B_2(B, N_a, A, N_b, T)$ can be instantiatiated to $B_2(B, N_a, A, N_b, \{N_a, N_b\})$ with the substitution $T \mapsto \{N_a, N_b\}$. Thus, we can optimize these rules to

$$
\begin{aligned}
&rule2,3: \ B_1(B, N_a, A) \to\\
&\qquad (\exists N_b)\ B_3(B, N_a, A, N_b, \{N_a, N_b\})\\
&\qquad\qquad M(B, A, \{N_a, N_b\}_{pk(A)}).
\end{aligned}
$$

Attack Preservation. In order to assure that our optimization technique is attack-preserving, we need to make further restrictions on the form of optimizable rules. For a pair (R, R') of rewrite rules to be combined, we require that the second rule has no messages on the left-hand side. We show with the help of a simplified example that allowing a message on the left-hand side of the second rule is unsafe.

Assume the following two rewrite rules for an agent in role B.

$$\begin{array}{l} rule1: \; B_0(B) \;\; \rightarrow B_1(B,B) \\ rule2: \; B_1(B,B) \;\, M(A,B,\mathit{sk(B)}) \rightarrow B_2(B,B) \end{array}$$

Since the state predicate $B_1(B,B)$ occurs in both $rule1$ and $rule2$ one might be tempted to optimize these rules to

$$rule1,2: \; B_0(B) \;\, M(A,B,\mathit{sk(B)}) \rightarrow B_2(B,B).$$

Assume furthermore that $M(A,B,\mathit{sk(B)})$ is an *impossible* message, because B never transmits *sk(B)*, and that B_1 is a state in which a protocol invariant fails, perhaps because it requires that the first two components of B's state must be different. Thus, the failure state is reachable in the original protocol specification, but since B_1 has been deleted by optimization and B_2 cannot be reached since M is impossible, the attack state is no longer present in the optimized protocol. This is why the left-hand side of the second rule is not allowed to have messages.

Name clashes. Before we can formally define the optimization of two rewrite rules, we have to deal with variable name clashes. In order to avoid accidentally introducing bindings between variables, we apply renaming functions. The following example illustrates the need for renaming.

Assume the following two rules which accidentally use the same variable X:

$$\begin{array}{l} rule1: \; B_0(B) \;\, M(X,B,A) \rightarrow B_1(B,A) \\ rule2: \; B_1(B,A) \;\; \rightarrow (\exists X) \;\, B_2(B,A,X). \end{array}$$

These rules are optimizable. The variable X is used in both rules, though there is no relation between the variable X of $rule1$ and the variable X of $rule2$. To avoid introducing a binding between these two independent variables, we rename X of $rule2$ to X' in the optimized rule. Thus, the optimized rule for $rule1$ and $rule2$ is

$$rule1,2: \; B_0(B)M(X,B,A) \rightarrow (\exists X') \;\, B_2(B,A,X').$$

The coincidence of variables B and A in the two rules is not a problem because the need to unify the B_1 facts in the two rules determines the appropriate substitution for them.

4.3 Optimization Steps

As intuitively illustrated in the previous section some restrictions on rules are necessary to guarantee an attack-preserving optimization. In summary, we only consider rules that describe asynchronously communicating, deterministically behaving agents, where each agent state is generated by at most one rule.

Local rule. For optimizations we deal only with local rules, in which only one state fact appears on the left and one on the right of the rule. The rules that arise from protocol transitions in an asynchronous environment are normally local, since only one agent changes state at a time.

A rule is *local* if it is of the form

$$R : F \ \mathbf{M} \rightarrow (\exists \mathbf{V}) \ F' \ \mathbf{M}'$$

where F, F' are state facts for the same role, and $\mathbf{M}$ and $\mathbf{M}'$ contain no state facts. Sets of variables and multisets of facts are denoted in bold-face letters.

Deterministic rule. States that are optimized away have to be deterministic in both directions. The first rule of an optimized pair needs a backward deterministic state on the right, while the second rule needs a forward deterministic state (the same state) on the left.

A state A_i is *forward deterministic* in $\mathcal{R}$ if there exists at most one rule in $\mathcal{R}$ with a fact of that state on its left-hand side. A local rule is forward deterministic if its state is forward deterministic.

A state A_i is *backward deterministic* in $\mathcal{R}$ if there exists at most one rule in $\mathcal{R}$ with a fact of that state on its right-hand side. A local rule is backward deterministic if its state is backward deterministic.

A state A_i is *deterministic* in $\mathcal{R}$ if it is both forward and backward deterministic.

Optimizable pair of rules. In the following definition of optimizable pairs of rules, $\mathit{Vars}(G)$ is the set of variables occuring in G.

Given a pair of local rules (R, R') in $\mathcal{R}$ of the form:

$$\begin{aligned}&R : F \ \mathbf{M_1} \rightarrow (\exists \mathbf{V_1}) \ G \ \mathbf{M_2}\\&R' : \ G' \rightarrow (\exists \mathbf{V_2}) \ H \ \mathbf{M_3}\end{aligned}$$

Then the pair (R, R') is *σ-optimizable* if

1. R and R' are local on the same role
2. There are no variable name clashes between R and R'
3. There exists a substitution σ on $\mathit{Vars}(G')$ such that $G'/\sigma = G$
4. The state of which G is a fact is deterministic.

As mentioned before, name clashes have to be resolved before our optimization technique is applied. Name clashes can always easily be resolved by renaming variables. In section 4.5 we describe how variable renaming can be efficiently realized for CIL.

Optimization step and optimized rule. We now can present the definition of the optimized rule for an optimizable pair of rules.

Given a pair (R, R') of σ-optimizable protocol rewrite rules of the form:

$$\begin{array}{l} R : F\ \mathbf{M_1} \rightarrow (\exists \mathbf{V_1})\ G\ \mathbf{M_2} \\ R' :\ G' \rightarrow (\exists \mathbf{V_2})\ H\ \mathbf{M_3} \end{array}$$

then an *optimization step* removes R, R' from $\mathcal{R}$ and replaces them with $R^o = \mathit{opt}(R, R')$, defined as

$$R^o : F\ \mathbf{M_1} \rightarrow (\exists \mathbf{V_1}\ \mathbf{V_2})\ \mathbf{M_2}\ (H\ \mathbf{M_3})/\sigma.$$

4.4 Properties of Optimization

We show that optimization is sound in the sense that it is attack-preserving. We also show that, under additional assumptions, it delivers a unique set of optimized rewrite rules regardless of the order in which optimization steps are applied. Detailed proofs can be found in [DMKFG00].

Protocol Security Invariants
Before we go into the details of the proof, we make some observations about protocol properties. Like Shmatikov and Stern [SS98], we only deal with protocol security properties that are invariants; that is, they are properties of the global state that are supposed to hold for all reachable states.

Furthermore, the invariants depend only on state facts and intruder memory, not on message facts. Secrecy invariants state that the intruder memory does not contain certain terms (which appear in the state memory of some honest principal), and other security properties such as agreements and precedence refer only to state facts. As pointed out in [SS98], if a security invariant is false, it remains false if the intruder's knowledge increases. They called this property *monotonicity of invariance.*

We make use of a more general characterization of security invariants. In protocols that can be expressed in CAPSL and translated to CIL, state memory is monotonic for honest principals as well. Once an honest agent holds a value for a protocol variable (associated with an argument position in its state memory), that value never changes. It follows that invariants that depend only on the global state cannot change their truth value once the relevant variables have become defined for a given agent. In particular, if a state invalidates a protocol invariant, every successor state will violate this invariant as well. We refer to this property as *persistence of violations.*

4.4.1 Soundness

Our soundness argument reasons about the *state graph* (S, T) of a rule set. The nodes of this state graph are the possible global states (multisets of ground facts) S of the protocol. The graph has directed, labelled transitions (edges) T consisting of pairs of states related by the instantiation of a rule, which labels the transition. A state is *reachable* if there is a sequence of transitions to it from the empty multiset, which is the initial state. We will refer to a state graph simply by its transition set T, since one can find all reachable states in it.

For example, the transition $s \overset{R/\alpha}{\to} s'$ means that there exists a rule $R : \mathbf{F} \to (\exists \mathbf{V})\ \mathbf{G}$ and a substitution α such that $\mathbf{F}/\alpha$ is a subset of s and the resulting system state s' is derived from s by replacing the multiset of ground terms $\mathbf{F}/\alpha$ with the multiset of ground terms $\mathbf{G}/\alpha$. (The substitution α assigns unused values to the variables in $\mathbf{V}$.)

Optimization steps eliminate two rules and replace them with a new combined rule. This changes the state graph by eliminating those transitions labelled with instantiations of the eliminated rules, and adding new transitions made possible by the new combined rule. The new state graph has the same set of states, but some of the states have become unreachable because some local states have been optimized away.

Attack preservation. An optimization step taking T to T' is *attack-preserving* if, for any security invariant φ, and any state s reachable in T that violates φ, there is a state s' reachable in T' that also violates φ.

Theorem 1 *Optimization steps are attack-preserving.*

In the proof of the above Theorem we make use of a lemma that shows that a transition instantiating the second rule R' of an optimizable pair (R, R') commutes with any other transition in T. The basic idea is to show that if a state violating a security invariant is reachable with a path that includes transitions due to one or both of the rules that have been eliminated by the optimization step, then that state is reachable using an alternate path that uses the new rule resulting from the optimization. The alternate path is constructed using commutativity properties implied by the lemma. Sometimes, one cannot reach the original violation state, but then one can reach another state reachable from the violation state, which must be a violation state by the persistence-of-violations assumption.

4.4.2 Termination and Uniqueness

The motivation for optimization is to reduce the number of states in order to speed up state evaluation tools such as model checkers. Our proposed optimization technique consists of single optimization steps performed in sequence. For analysis tools it is of importance whether the order in which optimization steps are performed has an impact on the final set of rules or on the final number of rules. We will show that optimization is terminating and delivers a unique result.

Optimization can be understood as a rewrite system in which a set of rules (a term) is rewritten to an optimized set of rules (another term). A well known result in the theory of rewrite systems says that a term has a unique normal form (i.e., it cannot be further rewritten into another term) if the rewrite system is canonical (for details see for instance [Sny91]). For a rewrite system to be canonical it has to be noetherian and confluent. Noetherian systems have no infinite sequences of rewrites. A rewrite system is termed confluent when for any term which can be rewritten into two different terms via several rewrite steps, there exists a common reduction term.

We show that our optimization process, understood as a rewrite system tranforming between sets of rules, is canonical. That means that optimization is a terminating process which delivers a unique set of rules as result. Therefore, speaking in terms of the associated state graphs, the original state graph T and the fully optimized state graph T', we can infer that T' is uniquely determined. For practical purposes that means that applying the optimization steps proposed in this paper in any order always leads to the same optimized state graph which can be used for security analysis.

Theorem 2 *Given a state graph T, then there exists a uniquely determined fully optimized state graph T'.*

In order to prove termination and uniqueness of optimization we make use of two lemmas. The first one states that a rule is optimizable with at most one rule to the right (in an optimizable pair) and at most one rule to the left.

Using this lemma, we can argue that a given set of rules can be arranged into a totally ordered list of rules such that two rules are optimizable only if they are adjacent (but adjacent rules do not necessarily form an optimizable pair). In the following we refer to such a list as list of optimizable rules. We show that optimization steps are locally confluent. That means one can always reach a common list of rules after two optimization steps (generally, local confluence allows for more than two rewrite steps).

The second lemma is concerned with local confluence. It states that if two optimization steps involve different optimizable pairs of rules, then they are commutative. That is, they might be executed in either order and the order has no effect on the resulting list of rules. If two optimization steps have a common rule, then after one optimization step the other optimization step is no longer applicable since the rule which both steps had in common has been deleted. But the new optimized rule can be taken for another optimization step to yield a common list of rules. Moreover, the optimization relation between rules is preserved. In summary, we show that performing optimization steps on a list of optimizable rules satisfies the following two conditions.

1. Performing an optimization step rewrites a list of optimizable rules into another list of optimizable rules. Assume in the original list the pair (R_1, R_2) has been optimized to R^o, then R^o is optimizable to

the left with whatever rule R_1 was optimizable to the left, and R^o is optimizable to the right with whatever rule R_2 was optimizable to the right.

2. Moreover, we show that optimization steps are locally confluent. That is, given a list of optimizable rules that can be rewritten into two different list of optimizable rules, we can always perform one more optimization step in order to reach a common list of optimizable rules.

Since the set of rules is finite and each optimization step reduces the number of rules by one, the optimization process is terminating. Therefore, the optimization process describes a rewrite system that is noetherian. A well known result in the theory of term rewrite system says that a system is canonical if it is noetherian and confluent. A noetherian system is confluent if and only if it is locally confluent. Thus, using the local confluence Lemma concludes the proof of Theorem 2.

4.5 Implementation

A CIL rewrite-rule optimizer has been implemented in Java and is applied as a post-processing step in the CAPSL-CIL translator. It is publicly available together with the CAPSL parser, type-checker and CAPSL-CIL translator at the CAPSL web site [Mil00b].

The optimizer starts reading a CIL specification and checks pairs of rewrite rules for optimizability. In order to decide whether two rules are optimizable, the optimizer needs to access information from the CIL specification. In particular, in order to decide whether two state facts are optimization compatible, the types of symbols are checked. This way we can guarantee that a proper substitution mapping between state predicates exists. Moreover, the optimizer needs to access assumptions and goals in order to check that the states to be eliminated are not named in goals and assumptions. As long as two optimizable rules are found, the optimizer computes the optimized rule, deletes the original rules and adds the new optimized one to the rule set.

Variable Renaming. In previous sections we mentioned the problem of name clashes. The simple-minded solution is to rename all variables in one of the rules in an optimizable pair to new variables. For instance, given the optimizable rules

$rule1:\ B_0(B,A)\ M(X,B,A) \rightarrow (\exists X) B_1(B,A)$
$rule2:\ B_1(B,A) \rightarrow (\exists X)\ B_2(B,A,X)$

we could rename the variables B, A, and X of $rule2$ using the renaming map $B \mapsto B', A \mapsto A', X \mapsto X'$:

$rule2:\ B_1(B',A') \rightarrow (\exists X')\ B_2(B',A',X')$

Now, the rules do not have any variable names in common and we may optimize them obtaining

$rule1,2:\ B_0(B)\ M(X,B,A) \rightarrow (\exists X^{'})\ B_2(B,A,X^{'})$.

As one can observe, some of the renamed variables are mapped back to their original name due to the given substitution map σ. For instance, in the example above B' has been mapped back to B using σ. Thus, a more efficient solution for eliminating name clashes is to only rename those variables which are not mapped by the substitution mapping σ.

Let (R, R') be an optimizable pair of protocol rewrite rules

$$R : F\ \mathbf{M_1} \rightarrow (\exists \mathbf{V_1})\ P_n(\bar{x})\ \mathbf{M_2}$$

and

$$R' : P_n(\bar{y}) \rightarrow (\exists \mathbf{V_2})\ G\ \mathbf{M_3}$$

where $F, G, P_n(\bar{x}), P_n(\bar{y})$ are state facts, $\bar{x} = x_1 \ldots x_r$, $\bar{y} = y_1 \ldots y_r$, and $\mathbf{M_1}, \mathbf{M_2}, \mathbf{M_3}$ are multisets of message facts. Let $\mathbf{W}$ and $\mathbf{W}'$ be the set of variables occurring in R and R' respectively.
The optimum of R and R', R^o, is computed by the following algorithm:

1. $\mathbf{E} = \{x_i \mid x_i = y_i \wedge i = 1, \ldots, r\}$
2. $\mathbf{C} = \{u \mid u \in \mathbf{W} \wedge u \in \mathbf{W}' \wedge u \notin \mathbf{E}\}$
3. $M_{ren} = \{u \mapsto u' \mid u \in \mathbf{C} \wedge u' \notin (\mathbf{W} \cup \mathbf{W}')\}$
4. $R_{ren} = R'/M_{ren}$

   Let $R_{ren} = P_n(\bar{y'})\ \rightarrow (\exists \mathbf{V_2'})\ G'\ \mathbf{M_3'}$
5. $M_{subst} = \{y_i' \mapsto x_i \mid y_i' \neq x_i \wedge i = 1, \ldots, r\}$

6. $R^o : F\ \mathbf{M_1}\ \rightarrow (\exists \mathbf{V_1}\ \mathbf{V_2'})\ \mathbf{M_2}\ (G'\ \mathbf{M_3'})/M_{subst})$

7. The symbol table of the CIL specification is updated with all newly introduced variables.

$\mathbf{C}$ represents the set of variables which may cause a clash. The variables in $\mathbf{C}$ are renamed in R' with new variables using the renaming map M_{ren}. The optimum is now computed using the new rule R_{ren}. At last, new variables are introduced in the symbol table.

We have applied the optimizer to several protocol specifications. The CAPSL specifications of the protocols may be found at our web site [Mil00b]. The CIL specifications were generated using the publicly available CAPSL-CIL translator. Table 4.1 shows the results. The reduction ratios clearly show

Protocol	*# Input Rules*	*# Output Rules*	*Reduction Ratio*
NSPK	7	5	28.57%
EKE	14	6	57.14%
Otway	9	6	33.33%
WMF	5	4	20%
SRP	19	8	57.89%
SSL	29	11	62.07%
Voucher	10	5	50%

Table 4.1: Reduction ratio of CIL rules

that the optimizer may reduce the number of rules significantly. This way, the performance of verification tools, such as finite-state exploration tools, can be drastically increased.

Chapter 5

Analysis Tools

Integration of CAPSL with protocol analysis tools includes two principal activities: development of connectors for interfaces from CIL to existing tools, and development of analysis techniques using general-purpose tools that can be adapted for this purpose. In particular, PVS can be applied to construct inductive proofs of protocols, and Maude can be used as a model checker with a suitable meta-level search strategy.

Connectors are being written to integrate CIL with a variety of formal security analysis tools, not only PVS and Maude, but also Athena [Son99, Mil00a]. In this section we describe the analysis techniques developed for the application of PVS and Maude. Before doing so, we summarize the common features of the connectors developed for PVS, Maude, and Athena.

5.1 Connector Design

Each connector has to fulfill the following two characteristics:

Syntactical and semantical correctness. One has to decide which CIL pieces are necessary for the translation into the targeted tool, and how they are translated. The resulting specification needs to meet all syntactical requirements of the target language and the translation has to express CIL constructs in a way that fits with the semantics of the analysis tool. In some cases CIL constructs can be translated in a straightforward fashion into concepts of the target language, in other cases the connector translation re-interprets CIL constructs and expresses them via appropriate (combinations of) language constructs of the target language.

Practicability. Though there might exists an obvious translation into the language of the analysis tool, the resulting specification might not be in a format that supports efficient and practical analysis. For instance, executability, non-determinism, and performance aspects of the transformed specification have to be taken into consideration. The connector algorithm might need to perform alterations or optimizations in order to meet these criteria.

In order to accomplish these objective, certain issues must be addressed:

- The translation strategy for transition rules
- Initialization
- Type and function limitations
- Goal generation
- Software engineering considerations for expediting connector writing.

The translation strategy is different for each tool, and generally it is not too difficult, due to the functional congruence between CIL rules and most tool transition rules.

Intialization refers to the transfer of protocol-specific declarations into the language of the analysis tool, as well as the generation of any protocol-specific attacker rules.

Tools are sometimes limited with regard to the set of datatypes and functions with which they can deal effectively. The connector must resolve incompatibilities between the tool's vocabulary and the typespecs provided in CAPSL by the prelude and the user.

The CAPSL translator passes goal statements on almost unchanged to the connector, which must do the best it can to generate tool-specific versions of those goals.

Given that we have written three connectors and expect that others will be written, we have attempted to make the common features as transferable as possible. There is fairly good support for writing connectors in Java. There is a Java class to parse the CIL output into a labelled tree structure, using another Java class defining the labelled tree type. The user-invoked connector class is typically a short standard program that invokes the parser and passes the parse tree to a workhorse translator class.

The parser and tree class can be re-used as is, and the user-invoked connector class of one connector is only a few lines different from that of another. The workhorse translator class is, of course, tool-dependent, but it can make good use of the tree-manipulating functions provided. Programming a connector in this environment is much like LISP programming.

5.2 PVS

The PVS specification and verification environment was developed at SRI and has been applied to many different design problems for high-assurance hardware and software [ORS92]. Our approach to using PVS for cryptographic protocol verification began with Paulson's trace model [Pau98]. We then modified and further developed the approach to work with a state-based model which is more compatible with the one built into CIL.

While the summary of our PVS approach in this section is intended to be readable without a prior background in the application of PVS, some experience with PVS would be necessary to put it into practice.

While the protocol modeling approach and the inductive proof technique apply to both secrecy and authentication goals, the main emphasis of this section lies in the description of our formalization of the secrecy theorem published in [MR00]. This theorem reduces secrecy proofs for protocols to first-order reasoning; in particular, discharging these proof obligations does not require any inductions. The trick is to confine the inductions to general, protocol-independent lemmas, so that the protocol-specific part of the proof is minimized. Moreover, secrecy protocols are modularized in the sense that there are separate verification conditions for each protocol rule.

We illustrate the encoding of specific protocols in our model using the Otway-Rees protocol [OR87]. We do not, however, go into details of proofs, since they are mostly straightforward adaptations of the ones stated in [MR00].

In order to formulate our results, we borrow the notion of ideals on strand spaces [THG98], and we show how this concept is useful in a state model context for stating and proving secrecy invariants. We show how the complement of an ideal, which we call a *coideal*, serves as a catalyst to apply Paulson's calculus-like set operators. Our protocol model is also unusual in that message events are interspersed with "spell" events that generate the short-term secrets in a session and specify which principals are supposed to share them.

Besides proving secrecy results of standard benchmark protocols like the Otway-Rees and the Needham-Schroeder (public key) protocols, our methods have been applied successfully (by B. Dutertre of SRI International) in the process of verifying the group management services of Enclaves [Gon97].

5.2.1 Modeling

The modeling task begins by defining the *primitive* data types that may occur as message fields: agents, keys, and nonces. This choice of primitive types is derived from Paulson's approach, and the PVS connector has to convert CAPSL types into these, such as from Principal (and all subtypes) to agent.

These sets of objects belonging to these primitive data types are assumed to be disjoint, and they are all subtypes (subsets) of the field type `field`. They are modeled as abstract datatypes in PVS.

An agent is either an 'ordinary' user, a dedicated server `Srv`, or the supposedly malicious `Spy`. Each agent `A` has some long-term keys: a public key `Pub(A)`, a corresponding private key `Prv(A)`, and a symmetric key `Shr(A)`.

Message fields are divided into primitive and compound fields. The primitive fields containing agents, nonces, and keys are constructed as `Agent(A)`, `Nonce(N)`, and `Key(K)`. (The PVS conversion mechanism is used to suppress these injections in the sequel.) Compound fields are constructed by concatenation or encryption. The concatenation of `X` and `Y` is the term `X ++ Y`. The encryption of `X` using the key `K` is `Encr(K,X)`, regardless of the type of key. The possible message fields are elements of the datatype `field`.

Agents and compound fields are never designated as secret by policy, though some compound fields may have to be protected to maintain the secrecy of some of their components. Thus, we define basic fields as nonces and keys, which are the kinds of primitive fields that may be designated as secret according to policy. The PVS definition of the membership predicate `basic?`

is shown below.

```
  basic?: set[field] = union(Nonce?, Key?)
```

As a notational convention, variables `A`, `B` and variants always stand for agents; `K` and variants always stand for keys; and `N` and variants are always nonces. `X`, `Y` and `Z` are arbitrary fields.

Each key `K` has an inverse.

```
 inv(K): key =
    CASES K OF
     Pub(A): Prv(A), Prv(A): Pub(A),
     Shr(A): Shr(A), Ssk(A): Ssk(A)
    ENDCASES
```

Thus, both `Shr(A)` and `Ssk(A)` are symmetric. The special agent `Server` is assumed to hold the symmetric (and thus, shared) key `Shr(A)` of any agent `A`.

There are three kinds of events: messages, spells, and state events.

```
   event: DATATYPE
   BEGIN
     Msg(Cont: field): Msg?
     Cast(Secrets: set[(basic?)], Cabal: set[agent]): Spell?
     State(Role: nat, Label: nat, Memory: field): State?
   END event
```

Messages are essentially Paulson's Says events, and the content of a message event is a field. We do not need to refer to the sender and receiver of a message. A *spell* generates certain session-specific primitive fields and designates them as secret. A spell is an event `Cast(S, C)`, where `S` is a set of short-term basic fields called the *book*, and `C`, the so-called *cabal*, is a set of agents who are permitted to share the secrets in `S`.

As a notational convention, we use `E` (and variants) to denote events, while `M` is a message and `C` is a spell.

A *global state* is simply a collection of events. Notationally, variants of `H` are global states. We shall see later that states reachable by a protocol contain messages in transit and local states of agents participating in the protocol.

```
global: TYPE = set[event]
```

We extend the notion of a content to global states in the natural way. Spells and state events do not contribute to the content. Similarly, the secrets of a state are obtained as the basic fields of the secrets of its cast events.

```
sees(H)(X): boolean =
  EXISTS (M: (Msg?)): member(M,H) & Cont(M) = X

secrets(H)(X): boolean =
  EXISTS (C: (Spell?)):
    member(C,H) & basic?(X) & member(X,Secrets(C))
```

5.2.2 Inductive Relations

The fundamental operations on sets `S` of message fields, as introduced by Paulson, are `Parts(S)`, `Analz(S)`, and `Synth(S)`.

Briefly, `Parts(S)` is the set of all subfields of fields in the set `S`, including components of concatenations and the plaintext of encryptions (but not the keys). Note that if `member(X, Parts({Y}))`, then `X` is a subterm of `Y`, in the sense of [THG98], written `X <= Y`. The subterm relation is a partial order.

`Analz(S)` is the subset of `Parts(S)` consisting of only those subfields that are accessible to an attacker. These include components of concatenations, and the plaintext of those encryptions where the inverse key is in `Analz(S)`.

```
Analz(S)(X): INDUCTIVE bool =
   S(X)
OR (EXISTS Y: Analz(S)(X ++ Y))
OR (EXISTS Y: Analz(S)(Y ++ X))
OR (EXISTS K: Analz(S)(Encr(K, X)) AND Analz(S)(inv(K)))
```

The intruder in our model synthesizes faked messages from analyzable parts of a set of available fields. This motivates the definition of `fake(S)`.

```
Fake(S): set[field] = Synth(Analz(S))

Fake_Parts: LEMMA Parts(Fake(S)) = union(Parts(S), Fake(S))
```

5.2.3 Ideals and Coideals

If the spy ever obtains some secret field `X`, it can transmit `X` as the content of a message. Thus, our secrecy policy is that if the message with content `X` occurs in some trace, then `NOT member(X,S)`, where `S` is a set of basic secrets.

The invariant that we will actually prove is that `NOT member(X, Ideal(S))`, where `Ideal(S)` is the *ideal* generated by `S`: the smallest set of fields that includes `S` and which is closed under concatenation with any fields and under encryption with keys whose inverses are not in `S`. Here, `Ideal(S)` is the k-ideal $I_k[S]$ from [THG98] where k is the set of keys whose inverses are not in `S`.

With our choice of k, the ideal is defined as follows:

```
Ideal(S)(X): INDUCTIVE boolean =
    S(X)
 OR (EXISTS Y, Z: X = Y ++ Z & (Ideal(S)(Y) OR Ideal(S)(Z)))
 OR (EXISTS Y, K: X = Encr(K, Y) & Ideal(S)(Y) & NOT S(inv(K)))
```

Under the assumption that any term not in the ideal may be already compromised, it is necessary to protect this whole ideal, because compromising any element of the ideal effectively compromises some element of `S`. It turns out that protecting this ideal is also sufficient.

The complement of and ideal, which we call a *coideal*, is denoted by `Coideal(S)`. This defines the set of fields that are *public* with respect to the basic secrets `S`, i.e., fields whose release would not compromise any secrets in `S`.

The property that makes the notion of "coideal" worth defining is that coideals are closed under attacker analysis, thereby implying that protection of the ideal is sufficient.

```
Analz_Closure: LEMMA Analz(Coideal(S)) = Coideal(S)
```

```
  Synth_Closure: LEMMA subset?(S,(basic?)) =>
                           Synth(Coideal(S)) = Coideal(S)
```

5.2.4 Protocols and Secrecy

A protocol specifies which messages or spells can be added to a global state. A secret in a spell book must be *unused* in the prior state, in the sense that it is not a part of any message content and it has not occurred as a secret in a prior spell.

```
  unused(H: global)(X: field): boolean =
     basic?(X) & NOT(Parts(sees(H))(X)) & NOT(secrets(H)(X))
```

A protocol rule is a triple consisting of a pre- and a post set of events and a set of nonces. Intuitively, such a rule is applicable in some global state `H` if the pre events are a subset of `H` and if the nonces in the rule are unused in `H`. A rule fires by deleting the pre events from the state and adding the post events.

```
 rule: TYPE =
    [# Pre: set[event],
       Nonces: set[(basic?)],
       Post: set[event] #]
```

There are several local conditions on protocol rules. First, there is at most one spell in the post, and a cast and a message event may not occur simultaneously in the post. Second, all secrets of casts in the post must be subset of the rule nonces. Third, regularity states that whenever a longterm key `K` is neither in the parts of the content or the memory of the pre then it is also not in the parts of the content or the memory of the post.

```
  single_spell(post: set[event]): boolean =
    FORALL (C, C1: (Cast?), E: (Event?)):
      (member(C, post) & member(C1, post) => C = C1)
    & (member(C, post) & member(E, post) => NOT Msg?(E))
```

```
fresh(Ns: set[(basic?)], post: set[event]): boolean =
  FORALL (C: (Cast?)): member(C,post) => subset?(Secrets(C),Ns)

regular(pre, tau1): boolean =
  FORALL(K: longterm):
    (NOT(Parts(sees(pre))(K)) & NOT(Parts(memory(pre))(K)))
       => (NOT(Parts(sees(post))(K))
          & NOT(Parts(memory(post))(K)))
```

It is usually straightforward to check that rules of a specific protocol obey these conditions. Usually, we (mis)use the PVS prover to automatically check these static conditions.

Rules that satisfies the conditions above are collected in the type `protocol`.

```
protrule(rl: rule): boolean =
   single_spell(Post(rl))
   & fresh(Nonces(rl),Post(rl))
   & regular(Pre(rl),Post(rl))

 protocol: TYPE = set[(protrule)]
```

A protocol `P` and a given set of initial knowledge `I` (of the spy), a *global* `I`-*extension* is a binary relation of states. This relation determines a transition system. An extension is either `honest`, i.e. it corresponds to a move by a player following the rules, or it is `faked` by the spy. As usually, the spy is reduced to add only messages with a content that can be inferred from the content of the current state and the initial knowledge.

```
honest(P: protocol)(H, H1): boolean =
  EXISTS(rl: (P)):
    subset?(Nonces(rl), unused(H))
    & subset?(Pre(rl), H)
    & H1 = union(Post(rl), difference(H, Prestates(rl)))

fake(I: set[field])(H, H1): boolean =
  EXISTS(X: (Fake(union(sees(H), I)))): H1 = add(Msg(X), H)
  global_extension(P: protocol, I: set[field])(H, H1): boolean
    = honest(P)(H, H1) OR fake(I)(H, H1)
```

We need some further concepts before stating our secrecy theorem. The *basic secrets* associated with a spell include not only the elements of the spell book but also the long-term secrets of the agents in the cabal.

```
  ltk(C: (Cast?))(X: field): boolean =
     Key?(X)
   & longterm(Val(X))
   & EXISTS(A: agent): Q(A)(Val(X)) & Cabal(C)(A)

  basic_secrets(C)(X: field): boolean =
     basic?(X) AND (Secrets(C)(X) OR ltk(C)(X))
```

A spell is *compatible* with an initial knowledge set that does not compromise its associated basic secrets, or mention the short-term secrets in its book.

```
  compatible(I: set[field])(C: (Cast?)): boolean =
    disjoint?(basic_secrets(C), Parts(I))
```

The set of reachable states `H` is defined in the usual way using a least fixed-point definition.

```
  reachable(P, I)(H): INDUCTIVE boolean =
     empty?(H) OR (EXISTS (G: global):
                    reachable(P, I)(G)
                    & global_extension(P, I)(G, H))
```

A protocol is secure with respect to its secrecy policy and the spy's initial knowledge `I` if every reachable state it generates is secret-secure. This property, for traces, was called "discreet" in [MR00].

```
  secret_secure(I: set[field])(H: global): boolean =
    FORALL C: compatible(I)(C) & H(C)
       => subset?(sees(H), Coideal(basic_secrets(C)))
```

The secrecy proof for a protocol has a protocol-independent part and a protocol-dependent part. The protocol-dependent part is expressed by the *occult*ness property defined below. It says that if the prior state is secret-secure, the next message event generated by the protocol does not compromise a secret. This has to be proved individually for each protocol. This protocol property was called "discreet" in [MR00].

```
occult(P: protocol): boolean =
  FORALL (I: set[field], H: global,
          C: (Cast?), rp: (protrule)):
       reachable(P, I)(H)
     & secret_secure(I)(H)
     & compatible(I)(C)
     & H(C)
     & subset?(Pre(rp), H)
     & P(rp)
     => subset?(sees(Post(rp)),Coideal(basic_secrets(C)))
```

The protocol-independent part of a secrecy proof is the Secrecy theorem. It only has to be proved once.

```
secrecy: THEOREM
  occult(P) => subset?(reachable(P, I), secret_secure(I))
```

The proof of this theorem is along the lines of the proof in [MR00] for proving a secrecy theorem for trace models, but now the induction is on the length of protocol extensions.

Notice that these are strictly secrecy results, and show only that the secrets generated in a particular run of the protocol are not compromised. Most authors of protocol proofs have noted that the security objectives of a protocol may be undermined in other ways than by compromising secrets, usually due to some failure of authentication. Possible combinations of secrecy and authentication are discussed in [MR00].

5.2.5 Example: The Otway-Rees Protocol

The goal of the Otway-Rees protocol is to mutually authenticate an initiator and responder and to distribute a session key generated by the server. The protocol consists of four messages, presented below as they appear in a CAPSL `MESSAGES` section.

The security objective is to prove that none of the secrets `Na`, `Nb`, or `K` are disclosed.

```
  or1. A -> B: M,A,B,{Na,M,A,B}Kas%F1 ;
  or2. B -> Srv: M,A,B,F1%{Na,M,A,B}Kas,{Nb,M,A,B}Kbs;
  or3. Srv -> B: M,{Na,Kab}Kas%F2,{Nb,Kab}Kbs;
  or4. B -> A: M,F2%{Na,Kab}Kas;
```

The full protocol specification includes `DENOTES` declarations indicating that `Kas` and `Kbs` are the keys shared by `A` and `B`, respectively, with the server `Srv`.

The PVS encoding of this protocol shown below was produced by hand. The PVS connector produces a much less readable version. The PVS connector is also, as of this writing, not yet capable of producing the spells and proof obligations automatically. Here we only state a selection of the formalization of the Otway-Rees protocol rules.

The spell rule `spl1` generates the nonce `Na` as needed for the first protocol step. Note that the server need not be mentioned in the cabal.

```
   spl1(A, B: agent, Na: nonce): (protrule) =
    (# Pre    := emptyset,
       Nonces := singleton(Na),
       Post   := singleton(Cast(add(Na,emptyset),
                                add(A, add(B, emptyset))))
     #)
```

The type constraint `(protrule)` causes the PVS type checker to generate verification conditions corresponding to the conditions on protocol rules. These and all the other verification conditions are easily discharged using the PVS prover.

Sending and receiving is split into two parts. The first step in the Otway-Rees protocol, for example, is transcribed as follows.

```
snd1(A, B: agent, N, Na: nonce): (protrule) =
    (# Pre    := add(State(roleA, 0, A ++ B ++ Srv),
                 add(Cast(add(Na, emptyset),
                 add(A, add(B, emptyset))), emptyset)),
       Nonces := add(N, emptyset),
       Post   := add(State(roleA, 1, A ++ B ++ Srv ++ Na),
                  add(Msg(N ++ A ++ B ++
                   Encr(Shr(A), Na ++ N ++ A ++ B)), emptyset))
     #)

rcv1(A, B: agent, N, Na: nonce): (protrule) =
    (# Pre    := add(State(roleB, 0, B ++ Srv),
                     singleton(Msg(N ++ A ++ B
                       ++ Encr(Shr(A), Na ++ N ++ A ++ B)))),
       Nonces := emptyset,
       Post   := singleton(State(roleB, 1, B ++ Srv ++ N ++ A))
     #)
```

Rules that introduce nonces (to be kept secret) take them from a prior spell with the expected cabal. When an agent uses a secret from a spell book, the agent does not see any of the other secrets in the same spellbook, though it might know about them from prior messages.

In general, a sequence of states generated by these rules interleaves the behavior of as many agents as we wish, and any number of concurrent or sequential sessions of the same agents. Altogether, the Otway-Rees protocol is formalized as follows.

```
 otway_rees: protocol =
    { r: (protrule) |
             EXISTS A, B, N, Na, Nb, K:
                    r = init(A, B)
                 OR r = spl1(A, B, Na)
                 OR r = snd1(A, B, N, Na)
                 OR r = rcv1(A, B, N, Na)
                 OR ...}
```

The secrecy theorem states that it suffices to show `occult(otway_rees)`. In a first step, using skolemization and split rules in order to show occultness for reach rule separately. For the lemma below occultness follows trivially for most protocol rules.

```
sufficient_for_occultness: LEMMA
    disjoint?(Msg?, Post(rp)) => occult(singleton(rp))
```

It remains to prove occultness for four rules in the Otway-Rees protocol. In the case of the `snd1` rule, for example, one has to prove:

```
{-1} subset?(sees(H), Coideal(basic_secrets(C)))
{-2} reachable(OR, I)(H)
{-3} H(C)
{-4} H(State(roleA, 0, A ++ B ++ Srv))
{-5} H(Cast(add(Nonce(Na), emptyset),
            add(A, add(B, emptyset))))
|----------
{1}  Coideal(basic_secrets(C))
       (N ++ A ++ B ++ Encr(Shr(A), Na ++ N ++ A ++ B))
```

Currently, we still prove these kinds of verification conditions in an interactive way (typically around 20-40 interactions per rule), but the repetitive patterns in these proofs suggest higher-level proof strategies.

5.2.6 Conclusions

Our secrecy theorem separates protocol-dependent and protocol-independent aspects of secrecy proofs. The protocol-dependent part is to show the occultness property, which only asks whether honest messages compromise secrets, given strong assumptions about the preservation of secrecy in the prior message history.

The secrets to be protected are defined in an explicit, uniform way by introducing "spell" events into the protocol. Spell events generate the short-term secrets for a particular "cabal", the set of agents sharing the new secrets. Secrets are shown to be protected even when the long-term secrets of other agents, or the short-term secrets in other protocol runs (with other spells) are compromised.

The closure results on the coideal have turned out to be a useful addition to the arsenal of proof techniques, enabling interesting examples to be shown secure. Protocol proofs are still complex enough so that we feel proof-checking and automation to be valuable for the sake of assurance, and we believe that the same techniques that simplify manual proofs will also be helpful in organizing machine-assisted proofs.

Currently, we are developing high-level PVS strategies for automatically discharging most verification conditions for typical protocol rules. In these strategies we try to capture the repetitive patterns that have been showing up in hand and mechanized interactive proofs. It is our hope that, using these strategies, we can prove secrecy results about realistic protocols in a fairly automatic way. We have developed an initial version of the PVS connector, but it needs to be improved to create more readable protocol theories and to incorporate goal information.

5.3 Maude

In this section we describe the design decisions and optimization solutions for a CIL connector to Maude. Maude is a novel, wide-spectrum executable formal specification language that has been successfully applied to communication and security protocols (see case studies on the Maude web page [Mau00]). In particular, Maude can be efficiently used as a model checker for security protocols as shown in [BD00, DMT00, DMT98b]. In order to minimize translation efforts into Maude and achieve maximal reusability of search strategies, attacker model and other predefined data structures, we designed a CIL-to-Maude connector that has been implemented in Java. Such a connector automatically translates CIL specifications into Maude, and, thus, enables a protocol designer to make use of the Maude model checking facilities without knowing Maude specifics. The following is a list of "ingredients" for the CIL connector to Maude:

- Specification of pre-defined CAPSL data types in Maude;
- Representation of the CIL model in Maude;
- Definition of an attacker model;
- Algorithm for translating protocol specific CIL constructs (such message lists and environments) into Maude;
- Definition of a general search strategy for model checking;
- Protocol-dependent and protocol-independent optimization techniques that improve the performance of the Maude model checker;

We illustrate the design of the CIL-to-Maude connector by means of an example. A prototypical implementation of the Maude connector in Java is finished. In the appendix we present the Maude specification that has been produced using the connector.

5.3.1 The Maude Language

Maude [CDE$^+$99, CELM96, Mau00] is a multi-paradigm specification language based on rewriting logic [Mes92]. Maude specifications can be efficiently executed using the Maude rewrite engine, thus allowing their use for system prototyping and the debugging of specifications.

Part of what makes Maude very well suited for the purpose of protocol analysis is its flexible wide-spectrum character: it can deal with very early design phases such as architectures and high-level designs, can be used to quickly develop executable prototypes, and can also be used to generate code. There is also a wide range of options on the kind of analyses that can be performed. One can develop formal models of a system very early, can debug formal specifications—which can be partial and incomplete—by executing them, can do more exhaustive model-checking and symbolic simulation analyses, or, for highly critical subsystems, can in fact do full formal verification using Maude's theorem proving tools.

The Maude model checker makes use of the reflective capabilities of rewriting logic and Maude [CM96]. Reflection allows user-defined execution strategies that can be formally specified by rewrite rules at the metalevel, including strategies such as breadth-first-search that can exhaustively explore all the executions of a system from a given initial state.

We briefly summarize the syntax of Maude. In the CIL-to-Maude connector we mainly make use of the following two types of modules:

- *functional* modules, that are equational theories used to specify algebraic data types; they are declared with the syntax `fmod ... endfm`, and
- *system* modules, that are rewrite theories specifying concurrent systems; they are declared with the syntax `mod ... endm`, and

Immediately after the module's keyword, the *name* of the module is given. After this, a list of imported submodules can be added. One can also declare *sorts* and *subsorts* (specifying sort inclusion) and *operators*. Operators are introduced with the `op` keyword. They are declared with the sorts of their arguments and result, and syntax is user definable using underscores '_' to mark the argument positions. Some operators can have equational *attributes*, such as `assoc` and `comm`, stating, for example, that the operator is associative and commutative. Such attributes are then used by the Maude engine to match terms modulo the declared axioms.

We make use of two kinds of logical axioms, namely, *equations*—introduced with the keywords `eq`, or, for conditional equations, `ceq`—and *rewrite rules*—introduced with the keywords `rl`, or for conditional rules `crl`. Functional modules can only have equations, whereas system modules can have any kind of axioms. The mathematical variables in such axioms are declared with the keywords `var` and `vars`.

5.3.2 Translation of the CAPSL Prelude

Our current connector is restricted to the operators provided in the CAPSL prelude (i.e., list concatenation, cryptographic operators for symmetric and public key encryption, etc.) which have been efficiently translated into Maude. The Maude module with the CAPSL prelude is automatically loaded with any CAPSL protocol specification.

In general any CAPSL type declaration can be translated into Maude. The reason for this restriction lies in the attacker model. The complexity of the attacker model is proportional to the number of function declarations and axioms. The more operators are defined, the more possible computations an attacker can perform. We restricted our attacker model to the usual functionality of composing and decomposing as well as encrypting and decrypting messages using the standard operators defined in the prelude (e.g., `cat,con,ped,se`). As a consequence, we only deal with type specifications from the CAPSL prelude.

Type, function, and constant declarations. Type and subtype declarations correspond to sort and subsort declarations in Maude. For instance, the declaration `TYPES Role, Spec, Agent: Object` translates into the Maude sort declarations

```
sorts Object Role Spec Agent .
```

and additional subsort declarations

```
subsorts Role Spec Agent < Object .
```

Since there exists no default supersort in Maude we have to explicitly define subsort relationship for all subtypes of `Atom`. Functions and constants are both translated into Maude operator declarations. In Maude an operator is defined by a name, a list of argument sorts and its target sort. Constants are operators with empty argument parameter. For instance, the CAPSL function `cat(Field, Field): Tape, ASSOC` is translated into `op cat : Field Field -> Tape [assoc]`. We introduced two additional boolean operators to deal with invertibility statements: `op INVERT_:_|_ : Field Field List[Field] -> Boolean` to capture invertibility axioms that have a list of fields as third parameter and `op INVERT_:_ : Field Field -> Boolean` to represent invertibility axioms with only two parameters.

Variables and axioms. Variable declarations such as `VARIABLES Al:Atom` are expressed in Maude as `var Al : Atom`. CAPSL axioms are represented by Maude (conditional) equations. For instance, the axiom `first(cat(Al, Xl)) = Al` looks like this `eq first(cat(Al,Xl)) = Al` in Maude. `IF-THEN-ELSE` axioms such as

```
IF keypair(PKl,PKIl) THEN ped(PKIl, ped(PKl, Xl)) = Xl
ENDIF
```

can be represented by a conditional equation

```
ceq ped(PKIl, ped(PKl, Xl)) = Xl
      if (keypair(PKl, PKIl) == true).
```

Properties. As for properties, associativity (`ASSOC`) and commutativity (`COMM`) are supported by Maude. The privacy property `PRIVATE` is treated indirectly. A private function symbol is one which cannot be accessed by the attacker (unless the symbol is private to the attacker). We provide a tailored solution to assure that the attacker only uses appropriate functions in rewrite rules. `CRYPTO` is treated similarly. So far we only handle the `FRESH` property of Nonces by introducing a counter that guarantees freshness of nonce values (see Section 5.3.4 and Section 5.3.5).

Using these general guidelines we translated the CAPSL prelude into a Maude module.

5.3.3 Definition of the CIL model

The flexibility of Maude allows us to simulate CIL rewrite rules using a syntactical representation that matches a mixfix[1] version of the CIL notation of Section 3.1.2. In order to achieve this goal, we provide a standard Maude specification that defines all sorts, operations and equations necessary to describe the specifics of a CIL model such as state, msg and intruder facts. The following functional theory (i.e., a functional module without equations) defines CIL facts.

```
fth FACT is
  protecting FIELDS .
  protecting CAPSLPRELUDE .

  sort Fact .

  op Msg : Principal Principal Fields -> Fact .   *** msg fact
  op State : Role MachineInt Fields -> Fact .   *** state fact
  op Net : Fields -> Fact .                    *** intruder fact
endfth
```

The imported module `FIELDS` defines an operator `op [_] : List[Field] -> Fields .` to represent field lists enclosed in square brackets such as `[A,B,Na]`. `LIST` is another parameterized module that defines list operations. The CAPSL prelude is also imported in order to refer to the sorts `Principal` and `Role`. Multiset of facts are then defined by the following theory.

```
fth FACTS is
  protecting MSET[Fact] .
  sort Facts .
  op [_] : MSet[Fact] -> Facts .

  op attack : -> Facts .
endfth
```

[1]We prefer to use the mixfix notation over prefix notation since it is more readable and shortens the protocol specifications. Thus, instead of using the prefix operator `facts` for multisets of facts, we used square brackets around multisets of facts. We also enclose term lists in square brackets.

The imported module MSET[Fact] defines multisets of facts using "," for separating facts. The term [State(roleA, 1, [A,B,Na]), Msg(UNK, A, [ped(pk(A), cat(Na,Nb))])] is of sort Facts due to the above theory (assuming appropriate variable declarations). This representation is very close to the CIL notation introduced in Section 3.1.2. The constant attack of sort Facts is used in the search. For a given protocol specification, we negate the goals and define systems states as patterns that invalidate the goals. We then define equations that rewrite a system state that invalidates a goal into attack. The search strategy terminates when an attack state is found. Further down we give details for the search strategy.

With the help of the above module we can mirror the CIL rules in Maude in a very straightforward way with only few alterations. One alteration is that Maude rewrite rules need a label. The current automated CIL-to-Maude translator numbers the rules consecutively.

5.3.4 Maude Attacker Model

The computational capabilities of an intruder are modelled by rewrite rules. Information that can be extracted from messages sent to, or intercepted by, the attacker are stored in the global state as data of the form Net(l) where l is a variable of sort Fields. We model possible attacker actions using additional rules that intercept messages (and place them on the net), fake new messages, decompose messages on the net and place their parts on the net, compose messages from parts, and encrypt messages. We give a few examples here.

```
crl [intercept] :
    [Msg(a,b,l), fs] => [Net(l), fs]
    if (not(Net(l) in fs) and not(spyOf(a) == true)) .
crl [fake] :
    [Net([a]), Net(l), fs]
    => [Net([a]), Net(l), Msg(Spy,a,l), fs]
    if (not(spyOf(a) == true) and not(Msg(Spy,a,l) in fs)) .
crl [fake2] :
    [Net([a]), fs] => [Net([a]), Msg(Spy,a,[a]), fs]
    if (not(spyOf(a) == true) and not(Msg(Spy,a,[a]) in fs)) .
crl [decompose] :
    [fs] => [analyze(nets(partition(fs))),
                    nonNets(partition(fs))]
    if (card(analyze(nets(partition(fs))),
                    nonNets(partition(fs))) > card(fs)) .
```

The `intercept` rule says that the content of any message that is not addressed to a spy can be intercepted by it (there may be more than one spy in the system) if the spy can learn some new data. The latter condition, formalized as `not(Net(l)`, assures that the intruder only intercepts when she gains some value by intercepting the message. If the intruder already knows the content of the message, that is `Net(l) in fs` (`fs` is a variable of sort fact set), than intercepting the message is useless. This condition speeds up the search process by avoiding exploring unnecessary states. `fake` states that for an agent `a` on the network and for a message `l`, the message `Msg(Spy,a,l)` is inserted into the global state. The condition assures that the spy does not send a faked messages to himself and that the message is new. Note that we have adopted the convention of [Pau98] where the message source and destination arguments are the true source and intended destination of the message, and they are not accessible to the receiver of the message. Hence, `Spy` is the sender here (and in `rcvMsg1` the sender is `unknown`). The rule `fake2` is similar but allows matching with only one state fact. The `decompose` rule is implemented in such a way that in one step the attacker retrieves the maximum amount of information from the currently held fields by applying decomposition and decryption functions defined in the prelude. The `decompose` rule partitions the fact set into attacker facts and non-attacker facts. The function `analyzed` recursively applies decomposition and decryption operators on the attacker facts until no more additional knowledge is extracted. For performance reasond, this rule only fires if the attacker knowledge can be increased. In order to achieve determinism for each composing step, the corresponding rules only create one new term using one of the composition or encryption functions defined in the prelude. Thus, for each of the constructional operators such as `ped, se, con, cat` there are rules in the Maude attacker module that describe state transitions that create new attacker facts. The attacker might need to apply these rules several times successively in order to build a composed message which he wants to fake.

5.3.5 Translating CIL Protocols and Environments

The parts of a CIL specification that are relevant for producing the corresponding Maude specification are Symbols, Axioms, Rules, Environment, and Goals.

Symbols

Symbol declarations are translated in very straightforward manner to Maude. Depending on the kind of symbol declaration, they result in Maude sort and subsort declarations or operator declarations.

Type declarations. A type declaration `symbol(`*ident1*,`type`,`ids()`,*ident2*, `props())` turns into sort declarations with associated subsort declarations: `sort` *ident1* and `subsort` *ident1* `<` *ident2*.

Constant or function declarations. The declaration `symbol(`*ident*, `op`, *args, val, props*) is translated into `op` *ident : args* `->` *val* `[` *props* `] .`

Variable declarations. A variable declaration `symbol(`*ident*, `var/pvar`, `ids()`, *val, props*) corresponds to the following Maude variable declaration `var` *ident* `:` *val* `.`

Properties of variables cannot be expressed directly in Maude. So far we treat `FRESH`, `CRYPTO`, and `EXPOSED` properties in the connector to Maude. They are not expressed by similar Maude concepts, rather we provide tailor-made solutions.

For the sort of a fresh variable we define constructors that fulfill the freshness requirement. For example in the case of nonces, our current implementation uses integers to enumerate instances of a sort. `op Nonce_ : MachineInt -> Nonce` defines a nonce generator that produces `Nonce 1, Nonce 2, ....` `MachineInt` is a Maude specific data type for integers. There are several ways to assure freshness of nonces. We chose to maintain a system counter as part of the protocol that is increased each time a fresh variable of some sort is generated.

Analogous to the `PRIVATE` property of functions, a `CRYPTO` property of a variable is expressed implicitly. That means that no attacker rule can make use of a crypto variable, unless it is held by the attacker or can be generated by the attacker using public functions. The semantics of the `RANDOM` property is still to be defined in CIL. `EXPOSED` terms will be handled in the initial state of an attacker. `EXPOSED(sk(Alice))` is translated into `Net([sk(Alice)])` expressing that the secret key of `Alice` is on the network (and thus, known to the attacker).

Axioms

It is relatively easy to translate CIL Axioms into Maude. CIL only uses equations, boolean predicates, if-then-else expressions and the invertibility predicate. Each of these concepts can the represented via conditional equations in Maude. For instance, the following set of CIL axioms

```
axioms(if(keypair(K1,K2),eqn(ped(K1,ped(K2,F)),F),true),
       keypair(sk(U),pk(U)),
       invertible(ped(pk(U),F), F, terms(sk(U)))
```

translates into the Maude (conditional) equations

```
ceq ped(K1,ped(K2,F)) = F if keypair(K1,K2) .
eq keypair(sk(U),pk(U)) = true .
eq INVERT ped(pk(U),F) : F | sk(U) = true .
```

Rules

Using the Maude module for the CIL model, we can mirror CIL rewrite rules almost identically in Maude. There are two main differences. For one, rules in Maude have to be labelled. We currently number rules rules consecutively. The second difference is that, depending on the solution we chose for generating fresh values, additional predicates might show up in the rules. This cannot be avoided, since the abstract concept of fresh values has to be implemented in an executable formalism like Maude. In the following we chose to use a counter fact `Cnt(n)`, which determines the next available, fresh integer in order to create nonces. The counter is updated any time a nonce is created. Though the counter is not by itself secret, and thus, it looks as if the attacker can generate any nonce (including those generated by honest agents), we assure in the attacker model, that the attacker does not access the counter, unless to create its own nonces.

The following rules correspond to the CIL rules for NSPK for sending and receiving the first message.

```
rl [1] :
   [State(roleA,0,[A,B]),Cnt(n),fs]
   => [State(roleA,1,[A,B,Nonce (n+1)]),
       Msg(A,B,[ped(pk(B),cat(Nonce (n+1),A))]),Cnt(n+1),fs] .

rl [2] :
   [State(roleB,0,[B]),Msg(UNK,B,[ped(pk(B),cat(Na,A))]),
    Cnt(n),fs]
   => [State(roleB,1,[B,Na,A]),Cnt(n),fs] .
```

`fs` is a free variable that matches the remaining facts in the multiset.

Conceptually, initialization rules could be translated one-to-one into Maude rules:

```
rl [initA] :
   [mt] => [State(roleA,0,[A,B])] .
```

The problem with such an initialization rule is that it is always enabled since `mt` is the identity element of the multiset operator `[_]` for `facts`, and thus, any system configuration has `mt` as a sub-configuration. This would unnecessarily cause the Maude model checker to loop infinitely. Moreover, the variables on the righthand side are free. The Maude model checker needs to bind variables to values in order to execute the protocol for specific agent instantiations. To resolve both problems, we decided to skip the initialization rules and instead provide a means of setting up agents for model checking sessions by using the environment information.

Environment

Each test environment results in an initial configuration that is modelled as a fact set. Currently, we only handle environment in which all agents exists concurrently. Here is the initial state for the test environment `Test1` of NSPK given in Section 2.5.

```
op Test1 : -> Facts .
```

```
eq Test1 = [State(roleA,0,[Alice,Bob]),
            State(roleB,0,[Bob]),
            Net([Mallory]),Net([Alice]), Net([Bob]),
            Net([ped(sk(Alice),Bob)]) ] .
```

The attacker knows the names of all principals that are part of the session and the exposed term.

Goals

The secrecy goal for `Na` in NSPK is localized into the CIL goal `loc( nodes( node(roleA,3), node(roleB,3)), secret(Na,ids(A,B)))`. This goal is violated for an agent `A`, if the `A`-agent is in a session with another honest agent, has the nonce `Na` in its memory, and the spy also knows this nonce (i.e., `Net([Na])` is in the multiset of facts). Therefore, we can characterize what states represent attacks. For the given NSPK secrecy goal, a state represents an attack when an agent has sent his nonce, encrypted, to another agent (different from the spy) but the nonce has been compromised by the spy, i.e., is on the net. For efficiency reasons, we implement this characterization in the following way. We search for the smallest `A`-state that contains the secret `Na`. For NSPK this is `State(roleA,1,[A,B,Na])`. Since the `A`-agent can grow over time, the state number can change and there might be additional slots after `Na`. Thus, we use the pattern `State(roleA,n,[A,B,Na,fl])` where the free variable `fl` matches the rest of the fields.

A violation of the secrecy goal for `Na` is defined by the following axiom.

```
ceq [Net([Na]),
     State(roleA,n,[A,B,Na,fl]), fs] = attack
    if not(spyOf(A) == true) and not(spyOf(B) == true) .
```

The predicate `spyOf` determines that the nonce was not deliberately sent by an agent to an intruder.

`PRECEDES` goals can also be interpreted in Maude with the help of additional equations. The `PRECEDES` goal `loc(nodes(node(roleA,3),node(roleB,3)), precedes(B,A,ids(Na)))` is fulfilled whenever `B` is in its final (third) state in which it holds a value for `A`, `B`, and `Na`, then there exists an agent in role `A` that agrees with `B` on these values. Here is the appropriate Maude formalization

```
ceq [State(roleB,3,[B,Na,A,Nb]),fs) = attack
    if not(State(roleA,n,[A,B,Na,fl]) in fs)
       and not(spyOf(A) == true) and not(spyOf(B) == true) .
```

5.3.6 Search Strategy and Optimization

The aim of model checking is to explore the state space for possible attacks. In problems like ours where the state space is infinite, model checking based on enumerative search constitutes a semi-decision procedure.

By applying rewrite rules, Maude can be used to build parts of the computation tree rooted at the initial state. In particular, the Maude interpreter delivers *one* particular branch of the computation tree determined by interpreter's default evaluation strategy for applying rules. However, to find possible attacks we need to explore *all* possible branches.

We proceed by employing Maude's metalevel reasoning capabilities and define, at the metalevel (in Maude), a *strategy* that specifies how the rules should be applied. To do this we define, within a Maude module, a function that implements an iterative deepening search on a tree specified, implicitly, by an initial state and rewrite rules of another module. Writing the search strategy for model-checking on the meta-level allows to import any protocol description and run the search strategy with that protocol. Thus, we specialize the search strategy to the given initial protocol state and the module defining the protocol rules. In doing so, the protocol specification becomes a term on the metalevel that is passed to, and manipulated by, the search strategy.

```
(fmod SEARCH is
  protecting META-LEVEL[NSPK] .
...
endfm)
```

For practicality reasons we defined a bounded depth-first search strategy instead of a breadth-first search strategy. A breadth-first search strategy can be obtained by iteratively calling the depth-first strategy with increasing depths. To specify a depth-first search strategy, we formalize a function whose arguments are the protocol module in which the reduction takes place, the current search path (i.e., a sequence of steps, where each step is a triple consisting of a rule label, a Maude-internal substitution number, and the new term), the current depth of the search tree, the maximum depths, and a list of protocol rule names (quoted identifier list `QIDL`). Initially, the path is the initial test term as defined by the environment specification `op bdfs :  Module Path MachineInt QidList -> Strategy .` `Qid` is a Maude specific data type for strings.

Below we show part of our bounded depth first seach strategy.

```
ceq bdfs(M,path(APATH,step(L,N,T)),DEPTH,MAX,QIDL)
    = if DEPTH < MAX
      then (if nextRewrite(M, T, L, QIDL) == none
            then backtrack(M,path(APATH,step(L,N,T)),
                           DEPTH,MAX,QIDL)
            else ids(M,path(path(APATH,step(L,N,T)),
                            nextRewrite(M,T,L,QIDL)),
                     DEPTH + 1,MAX,QIDL)
            fi)
      else backtrack(M,path(APATH,step(L,N,T)),DEPTH,MAX,QIDL)
      fi
    if T =/= {'attack}'Facts .

ceq bdfs(M, path(APATH, step(L, N, T)), DEPTH, MAX, QIDL)
    = stop(DEPTH, path(APATH, step(L, N, T)))
    if T == {'attack}'Facts .
```

The strategy conceptually builds a search tree, where each node corresponds to a state of the protocol, and each successor node is reached by applying a rewrite rule, matching the variables of the rule with values from the current multiset of facts. The search strategy remembers the current path of the search tree as a sequence of steps. A step records the latest rule that has been applied, the substitution number (that is a Maude specific number from which one can derive the substitution between the protocol state and the rule variables) and the new protocol state (`op step :  Qid MachineInt Term -> Step`).

For a given state, this strategy performs a rewrite step that generates a successor state. At each step, we test for an attack or backtrack in case the branch terminates or the maximum depth is reached.

`{'attack}'Facts` is the representation of the NSPK term `attack` at the search meta-level.

In theory, iterative deepening can find any attack, but in practice heuristics are needed to do so using manageable resources. The performance of a search strategy like the one above can be enhanced significantly if one adds heuristics that tune the model checking process to the application at hand. In the case of security protocols analysis we have gained lots of experience by model checking several protocols. In the following we summarize our main ideas to speed up the model checking process.

Protocol-independent Optimization

Which rules and in what order those rules are applied has an impact on the performance of the model checker. Our optimization techniques and heuristics for the Maude model checker either effectively prune the search tree or reorder it.

Discarding rules. Some rules might be redundant in the sense that they are not necessary to find attacks. For instance, the overhear rule of an the attacker is an example of such a rule. As long as there are rules for intercepting and faking a message, the effect of overhearing a rule can always be simulated by replaying it. Therefore, we chose not to implement such a rule in our attacker model.

Prioritizing rules. During the process of building the search tree one can apply re-ordering functions that give priorities to rules which are expected to lead faster to attacks. This is a heuristic in the sense that for different protocols and attacks different heuristics might turn out to be better. In the protocols we investigated we were successful with ordering the Maude rewrite rules, such that the attacker intercept rule gets high priority, next followed by the rules which describe the protocol, and lower priorities for the other attacker rules, with the composition rules having the lowest priority. Essentially we followed the intuition that, the more restricted the enabling condition for a rule is, the less likely it is that it will occur. Thus, giving it a high priority does not result in search trees, in which those rules are always applied first. This is similar to optimization techniques suggested by Shmatikov et al [SS98]. In their approach an intruder always intercepts (our weaker version: intercept has high priority), and an intruder does not send if an honest agent can send (our version: protocol rules have higher priority then fake rules).

Restricting rules sequences. Another optimization technique we used was to prune the search tree such that each sequence of applied rules satisfies special successor-rule conditions. For example, once an attacker fakes a message, he does that with the intent of some agent receiving that message. Thus, during the search we assure that a fake rule is always followed by a rule which denotes the receipt of the message. Moreover, optimization steps as discussed in Chapter 4 can also be implemented on the meta-level of search. In [BD00] we formalize that certain action sequences must occur as a block (without other interleaved actions). For example, local computations of agents, such as receiving a message, followed by internal computations, followed by sending a new message, can be summarized to one step. Similarly, a message which was intercepted by an attacker should be decomposed in the subsequent step. One can define arbitrary dependencies between rules and enforce their order in the search strategy.

As long as those dependencies are only used to reorder the search tree, there is no danger of missing out on an attack. In those cases where one can show that the optimization preserves all attacks, one can even prune the search tree.

The above listed optimization ideas have been implemented in Maude on the meta-level,and,thus, they are independent of the protocol to which the search is applied. Further details can be found in [BD00].

Protocol-dependent Optimization

Other optimization techniques depend heavily on the protocol which is to be analyzed. We made use of two techniques.

Message format This technique tests the message format of composed and faked messages. For this purpose we added conditions to the fake-rule and the rules for composing (i.e., concatenation or encryption) such that they are only enabled if the resulting field fulfills the format of the message contents of the protocol. We defined a predicate `isInMsgContentFormat` that is defined to be true for any field that has (partially) the format of a valid protocol message. The following describes part of the definition for NSPK.

```
eq Net([A,Na]) isInMsgContentFormat = true .
eq Net([ped(K,cat(Na,A))]) isInMsgContentFormat = true .
```

Receivability of faked messages. Moreover, we test whether a faked message is receivable in a given state of the protocol. For instance, the first message of NSPK is receivable whenever an agent in role `B` in state zero exists in the multiset of facts.

```
eq Msg(UNK,b,[ped(pk(B),cat(Na,A))]) isReceivableIn
   [State(roleB,0,[B]), fs] = true .
```

5.3.7 Conclusion

In the process of designing a CIL connector to Maude, we tackled some essential issues about the practicability of a connector. Our aim is not just to translate the CIL specification into an executable Maude specification, but to yield an efficiently executable and practically analyzable protocol specification. In order to meet this goal, we solved the issues involved in translating CIL into an equivalent Maude specification and we proposed and fine-tuned several optimization techniques that will improve the performance of the Maude model checking tool.

A prototype of the CIL-to-Maude connector has been implemented in Java using the existing support classes. The implementation of the CIL-to-Maude connector took one-person week.

As mentioned before the current connector is restricted on the predefined data types of the CAPSL prelude and supports the optimization strategies discussed in the previous session. Order specification in environment declarations are not yet handled. In environment specifications one can define a principal as exposed, meaning that all the secrets of that principal are known to the attacker. This issues needs to be addressed in future extensions of the connector. Further investigation is also necessary in order to define the semantics of protocol goals other than secrecy or agreement goals.

5.4 Athena

Athena is a model checker for security protocols [Son99], based on the strand space representation [THG98]. The required input format for Athena was obtained from draft material supplied by its author, Song.

An Athena specification has two parts: a sequence of strands and a sequence of verification conditions. A strand is a sequence of nodes. A node consists of a "sign" or direction (send or receive, represented by "->" and "<-") and a term representing the content of a message. Nonces introduced or "originated" in a sent message are also listed. A strand specification of a protocol is a *parametric strand* in the sense of [CDL+00], which addresses translations between MSR and strand space protocol models in a general setting. A strand specification is parameterized by the list of protocol variables that must be instantiated to create a particular strand.

Here is the "A" strand from the NSPK example in Athena:

```
  P_A(0,3) {
  VAR: P_A, P_B, NONCE_Na, NONCE_Nb;
  -> : E{C[NONCE_Na,P_A],PUBKEY_P_B}
       | New(NONCE_Na);
  <- : E{C[NONCE_Na,NONCE_Nb],PUBKEY_P_A};
  -> : E{NONCE_Nb,PUBKEY_P_B};
}
```

Issues in writing a connector to Athena come up in five areas:

1. The basic translation strategy to produce strands
2. Normalization: non-message rules
3. Type and function limitations
4. Goal generation

5.4.1 The Translation Strategy

CIL rewrite rules update the state of exactly one role at a time. A rule may have a received message on the left or a sent message on the right, or both.

A message on the left generates a "receive" node in the strand associated with the rule's role, and a message on the right generates a "send" node for the same role. Any variables listed as nonces of the rule become nonces of the send node. In [CDL$^+$00], MSR specifications were "normalized" to generate all nonces in the initialization rule of each role, but we do not do that, because we would lose the information as to which node originated the nonce, which is required for Athena.

Rewrite rules are not required to be in the expected order associated with message events or state changes, and the connector was written so that nodes will be added in the correct sequence regardless of the rule order. However, the CAPSL translator now generates rules in the expected order, primarily to make the rule output as readable as possible, and future connectors should be able to take advantage of that.

There must be a strand for each role in the protocol. To find the roles, we can get a list of symbols of type Role from the CIL symbol table; but this not quite right because the translator generates a role constant for every variable of type Principal. Such a variable is usually a role, but it could be just a message field, and the translator does not check whether a message is actually sent to or from that principal. The connector generates an empty strand for a non-role principal, which is a nuisance but not a serious problem. The CAPSL translator should probably refrain from generating roles for such variables.

The strand parameters are the protocol variables that must be instantiated to produce a particular strand. These variables occur as slots in states of the strand's role, and they are found in the slot table.

Once the protocol variable parameters used by the strand are found, we have to ensure that these same variables are used in the strand node specifications. Strands are generated from rewrite rules, but, as remarked earlier, the variables in rewrite rules are, in principle, dummy variables that are subject to renaming, and the renaming can be different in each rule. The CAPSL translator ordinarily uses the original protocol variables in the rules, for the sake of rule readability, but there is presently no guarantee of that. Hence the connector replaces the rule variables with protocol variables. This is done using the slot table. The protocol variable corresponding to a rule variable is found by observing which slot it occupies in the rule's right-hand state fact, which should have them all.

In an Athena specification, variable names are prefixed by the variable type name. The mapping of CAPSL type names to corresponding Athena type names is built into the present connector. This topic is discussed further below under the limitations issue.

5.4.2 Normalization: Non-Message Rules

A normalized MSR rule in [CDL+00] either sends or receives one message. CIL rules may have a message on both the receive and the send side, and they could also have no messages, as in the case of state initialization rules, actions that assign a value to a new variable, and actions that test an equation or other boolean condition.

Initialization rules are not a problem; they just create a strand to which nodes will be added. Rules that both send and receive messages are also no problem, they just create two nodes. Non-initialization rules with no messages are a problem because no node is created for them; the information they carry is lost.

An assignment action in CAPSL, such as $C = A$, generates a rule like $B_1(B, A) \rightarrow B_2(B, A, A)$ and creates a slot table entry for C as slot 3 of B. Later rules will refer to slot 3 of B by the variable C without regard for its value. For example, the message $B \rightarrow A : C$ becomes $B_2(B, A, C) \rightarrow B_3(B, A, C), M(B, A, C)$. The B strand will have A, B and C as parameters, and the connector will give it a node to send C. The correct strand should send A instead.

Fortunately, the assignment action problem goes away because the CAPSL optimizer combines assignment rules with other rules, so that the CIL output only has the single rule $B_1(B, A) \rightarrow B_3(B, A, A), M(B, A, A)$, which generates the correct node. This may be viewed as a kind of normalization step.

We are less fortunate with test actions. A test $C = A$ (in a state where C and A are held) generates two rules, one to evaluate the equation and one to continue if it is true: $B_2(B, A, C) \rightarrow B_3(B, A, C, C = A)$ and $B_3(B, A, C, \texttt{true}) \rightarrow B_3(B, A, C)$. Neither of these generates a node, and the information that $C = A$ is lost. Both C and A are strand parameters, and they could be instantiated to be unequal to produce a strand that is inconsistent with the protocol. Furthermore, the optimizer does not help here; it can only combine the second rule with a later one.

The way to handle this, at the moment, is just to write the protocol without putting in test actions. Most protocols don't need them; their main purpose in CAPSL is to control conditional branching, an advanced feature.

5.4.3 Type and Function Limitations

CAPSL permits new field types, as well as new encryption and other computational functions, to be introduced using abstract datatype specifications (called *typespecs* in CAPSL). Analysis tools are often limited in their adaptability to extensions. Like the PVS tools we are developing, Athena can presently deal only with simple abstract operators. The ones currently supported are public-key and symmetric-key encryption, concatenation, hashing, MAC (keyed hash) and a few standard data types like `P` for principals and `NONCE` for nonces. The connector translates the equivalent CAPSL standard types and functions (in the prelude) into them – for example, Principal to `P`, `ped`(.,.) to `E`{.,.}, etc.

For types and functions without equivalents in Athena, the connector preserves their names, and they appear in the connector output. The Athena user can then see the unrecognized symbols and attempt to rewrite the protocol specification without them. The connector can be modified easily to add more types and functions. When future versions of Athena permit user-defined new field types and functions, the connector will have to be extended to make use of the symbol table and axiom entries in the CIL.

5.4.4 Goal Generation

Goals in Athena are specified with a list of partially instantiated strands, with conditions as to which combinations of these strands are permitted to appear in a bundle. A secrecy goal says that a strand with a given value for a secret variable is not compatible with a standard intruder (or "penetrator") "flush" strand with the same value. An agreement goal says that the existence of a strand with values for certain variables implies the existence (in the same bundle) of another strand with the same values for the same variables. These goals can be generated in a straightforward way from the `SECRET` and `PRECEDES` goals in CAPSL.

In the current version of Athena, symbolic constants used to indicate values of variables are always formed simply by appending "0" to the name of the variable, and the connector does that. More general value assignments are possible in CAPSL through the use of environment modules.

As an example, the CAPSL goal appearing in CIL as `precedes(B,A,ids(Na))` says that if A reaches its final state, there is or was a state of B agreeing with A on B, A, and N_a. This goal is stated in Athena as:

```
VC. {Strand(0,3)[(P_B,B0),(P_A,A0),
                 (NONCE_Na,Na0)]} =>
    {Strand(1,3)[(P_B,B0),(P_A,A0),
                 (NONCE_Na,Na0)]}
```

where strand 0 is the A strand and strand 1 is the B strand.

Chapter 6

Concluding Remarks

CAPSL, CIL and the translation between them are designed to address important goals in cryptographic protocol specification for analysis purposes. With a common specification language, it becomes possible to harness the combined power of many tools for protocol analysis in a practical way. The components of the CAPSL environment include transportable software for translation of CAPSL to CIL, and connectors to adapt CIL to the input languages of various analysis tools. This software is still under development, but a CAPSL-to-CIL translator is available on the Web site `http://www.csl.sri.com/~millen/capsl`. The site also contains more examples of CAPSL specifications and other documentation.

With CAPSL, one can express protocols in the simplest accepted message-list form. Type specifications in CAPSL and their use for introducing new operators and subtypes bring an expanding class of protocols within reach. There are plans to broaden the applicability of CAPSL further with extensions for multicast protocols.

CAPSL simplifies what used to be the most awkward aspect of abstract protocol specification, the distinction between short-term session data and the long-term data associated with persistent entities. This was done by applying the general type specification mechanism, together with the novel concepts of private functions and invertibility axioms.

The intermediate language CIL was chosen with an eye toward a clear analysis-level modeling semantics and a universal pattern-matching transition rule style that lends itself both to model checking and inductive proof techniques.

We have techniques for inductive protocol proof using PVS and model checking using Maude. In the process, we have confirmed that CIL output is a good match for the specification needs of these tools. With the Athena connector, we have made a start on linking CAPSL to independently developed analysis tools as well.

Bibliography

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

[BD00] D. Basin and G. Denker. Maude versus Haskell: an Experimental Comparison in Security Protocol Analysis. In K. Futatsugi, editor, *Third Intern. Workshop on Rewriting Logic and Its Applications, Kanazawa City Cultural Hall, Kanazawa, Japan, September 18-20, 2000*, pages 235–256. *To appear:* Elsevier Science B.V., Electronic Notes in Theoretical Computer Science, `http://www.elsevier.nl/locate/entcs/`, 2000.

[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)*, 1999.

[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.

[CDE$^+$99] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. Quesada. *Maude: Specification and Programming in Rewriting Logic*. SRI International, Computer Science Laboratory, Menlo Park, CA, January 1999. `http://maude.csl.sri.com/manual/`.

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

[CDL$^+$00] I. Cervesato, N. Durgin, P. Lincoln, J. Mitchell, and A. Scedrov. Relating strands and multiset rewriting for security protocol analysis. In *13th IEEE Computer Security Foundations Workshop*. IEEE Computer Society, 2000.

[CELM96] M. Clavel, S. Eker, P. Lincoln, and J. Meseguer. Principles of Maude. In Meseguer [Mes96], pages 65–89.

[CJM98] E. Clarke, S. Jha, and W. Marrero. Using state space exploration and a natural deduction style message derivation engine to verify security protocols. In *Proc. IFIP Working Conference on Programming Concepts and Methods (PROCOMET)*, 1998.

[CM96] M. Clavel and J. Meseguer. Reflection and Strategies in Rewriting Logic. In Meseguer [Mes96], pages 125–147.

[DM99a] G. Denker and J. Millen. CAPSL and CIL Language Design: A Common Authentication Protocol Specification Language and Its Intermediate Language. CSL Report SRI-CSL-99-02, Computer Science Laboratory, SRI International, Menlo Park, CA 94025, 1999. `http://www.csl.sri.com/~denker/pub_99.html`.

[DM99b] G. Denker and J. Millen. CAPSL intermediate language. In *FLoC Workshop on Formal Methods and Security Protocols*, 1999.

[DM00] G. Denker and J. Millen. CAPSL integrated protocol environment. In *DARPA Information Survivability Conference (DISCEX 2000)*, pages 207–221. IEEE Computer Society, 2000.

[DMKFG00] G. Denker, J. Millen, J. Kuester-Filipe, and A. Grau. Optimizing protocol rewrite rules of CIL specifications. In *13th IEEE Computer Security Foundations Workshop*, pages 52–62. IEEE Computer Society, 2000.

[DMT98a] G. Denker, J. Meseguer, and C. Talcott. Protocol specification and analysis in Maude. In *Formal Methods and Security Protocols*, 1998. LICS '98 Workshop.

[DMT98b] 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`.

[DMT00] G. Denker, J. Meseguer, and C. Talcott. Formal Specification and Analysis of Active Networks and Communication Protocols: The Maude Experience. In D. Maughan, G. Koob, and S. Saydjari, editors, *Proc. DARPA Information Survivability Conference and Exposition, DISCEX2000, January 25-27, Hilton Head Island, SC, USA*, pages 251–266, 2000. `http://schafercorp-ballston.com/discex/`.

[GNY90] L. Gong, R. Needham, and R. Yahalom. Reasoning about belief in cryptographic protocols. In *IEEE Symposium on Research in Security and Privacy*, pages 234–248. IEEE Computer Society, 1990.

[Gon97] L. Gong. Enclaves: enabling secure collaboration over the Internet. *IEEE J. of Selected Areas in Communications*, 15(3):567–575, April 1997.

[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.

[Mau00] Maude Web Site. `http://maude.csl.sri.com/`, 2000.

[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.

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

[Mes96] J. Meseguer, editor. *Rewriting Logic and Its Applications, First International Workshop, Asilomar Conference Center, Pacific Grove, CA, September 3-6, 1996.* Elsevier Science B.V., Electronic Notes in Theoretical Computer Science, Volume 4, `http://www.elsevier.nl/locate/entcs/volume4.html`, 1996.

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

[Mil00a] J. Millen. A CAPSL connector to Athena. In H. Veith, N. Heintze, and E. Clarke, editors, *Workshop of Formal Methods and Computer Security.* CAV, 2000.

[Mil00b] J. Millen. CAPSL web site. `http://www.csl.sri.com/~millen/capsl`, 2000.

[MR00] J. Millen and H. Rueß. Protocol-independent secrecy. In *2000 IEEE Symposium on Security and Privacy.* IEEE Computer Society, 2000.

[NS78] R. Needham and M. Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21(12):993–998, December 1978.

[OR87] D. Otway and O. Rees. Efficient and timely mutual authentication. *ACM Operating System Review*, 21(1):8–10, 1987.

[ORS92] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752. Springer-Verlag, 1992.

[Pau98] L. Paulson. The inductive approach to verifying cryptographic protocols. *Journal of Computer Security*, 6(1):85–128, 1998.

[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.

[Sny91] W. Snyder. *A Proof Theory for General Unification.* Birkhäuser, 1991.

[Son99] D. Song. Athena: a new efficient automatic checker for security protocol analysis. In *12th IEEE Computer Security Foundations Workshop*, pages 192–202. IEEE Computer Society, 1999.

[SS98] V. Shmatikov and U. Stern. Efficient Finite State Analysis for Large Security Protocols. In *11th IEEE Computer Security Foundations Workshop, Rockport, Massachusetts, June 1998*, pages 106–115. IEEE Computer Society, 1998.

[THG98] J. Thayer, J. Herzog, and J. Guttman. Honest ideals on strand spaces. In *11th IEEE Computer Security Foundations Workshop*, pages 66–78. IEEE Computer Society, 1998.

Appendix A

CAPSL and CIL Syntax

A.1 CAPSL Syntax

Here is an informal presentation of the CAPSL concrete syntax. In this grammar, curly brackets { } indicate a sequence of one or more of the enclosed item. A vertical bar — separates choices. Optional items are enclosed in square brackets [ ]. Literal tokens appear enclosed in single quotes ', except for keywords, which are all caps.

There is a general meta-rule for forming nonterminals representing lists. If x is a nonterminal symbol, then x`_list` represents zero or more occurrences of x separated by commas.

Comments in CAPSL are surrounded by `/* */`, e.g., `/* this is a comment */`.

The grammar permits constructs that are illegal for semantic reasons, such as improper ordering or type inconsistency. The grammar also permits some illegal constructs that could have been eliminated with a more elaborate grammar, but can also be handled by later checks. An example is that the % operator should be used only in message fields.

Identifiers are sequences of alphabetic characters and digits, and may also contain the underline _ character. An identifier that consists solely of digits is a number.

```
specification:
   {protocol | typespec | environment}

protocol:
   PROTOCOL ident ‘;’
   {declaration}
   [ASSUMPTIONS
        {assertion ‘;’}]
   MESSAGES
      phrase_seq
   [GOALS
        {assertion ‘;’}]
   END;

typespec:
   TYPESPEC ident ‘;’
   {declaration}
   AXIOMS {statement ‘;’}
   END;

environment:
   ENVIRONMENT ident ‘;’
   {declaration}
   [AXIOMS
      {statement ‘;’}]
   {agent}
   [EXPOSED term_list ‘;’]
   [ORDER order ‘;’]
   END;

agent:
   AGENT ident HOLDS
      {equation ‘;’}

order:
   ident            /* of agent */
   |
   ‘(’ order ‘;’ order ‘)’
   |
   ‘(’ order ‘||’ order ‘)’
```

```
declaration:
   IMPORTS ident_list ‘;’
   |
   FUNCTIONS {func_dec}
   |
   VARIABLES {variable_dec}
   |
   CONSTANTS {variable_dec}
   |
   DENOTES {equation [‘:’ ident_list] ‘;’}
   |
   TYPES {type_dec}

assertion:
   HOLDS ident ‘:’ ident_list
   |
   BELIEVES ident ‘:’ assertion
   |
   KNOWS ident ‘:’ assertion
   |
   ASSUME assertion     /* as an action */
   |
   PROVE assertion      /* as an action */
   |
   SECRET ident [‘:’ ident_list]
   |
   AGREE ident_list ‘:’ ident_list ‘|’ ident_list
   |
   PRECEDES ident ‘:’ ident ‘|’ ident_list
   |
   statement

statement:
   logicstmt
   |
   IF logicstmt THEN simplestmt [ELSE simplestmt] ENDIF
   |
   INVERT term ‘:’ ident | term_list

logicstmt:
   simplestmt
   |
   NOT ‘(’ simplestmt ‘)’

simplestmt:
   equation
   |
   term

equation:
   term ‘=’ term

variable_dec:
   ident_list ‘:’ ident [‘,’ property_list] ‘;’

type_dec:
   ident_list [‘:’ ident] ‘;’

func_dec:
   ident ‘(’ ident_list ‘)’ ‘:’ ident [‘,’ property_list] ‘;’

phrase_seq:
        phrase
        |
        phrase [‘/’] phrase_seq

phrase:
   [{action ‘;’}]
   message
   [{action ‘;’}]
   |
   invocation
   |
   selection

action:
   equation | ASSUME assertion | PROVE assertion

invocation:
   INCLUDE ident ‘;’    /* naming a protocol */
```

```
selection:
   IF statement THEN phrase ELSE phrase ENDIF ‘;’

message:
   [ident ‘.’] ident ‘->’ ident ‘:’ term_list ‘;’

property:
   CRYPTO | FRESH | PRIVATE | EXPOSED | ASSOC | COMM

term:
   ident
   |
   functioncall
   |
   bracket
   |
   lowe
   |
   paren

functioncall :
   ident ‘(’ term_list ‘)’

/* arithmetic expressions with +, -, *, /, and ^
(for exponentiation) are supported, and
treated as function calls on pls, mns, etc. */

bracket:
   ‘{’ term_list ‘}’ [‘’’] [term] /* single quote for decrypt */
   |
   ‘[’ term_list ‘]’ [‘’’] [term]

lowe:
   term ‘%’ term

paren:
   ‘(’ term ‘)’
```

A.2 CIL Syntax

This is the syntax of CIL output in functional notation. Identifiers in quotes are literal tokens, as are parentheses and commas. Other identifiers are nonterminals. We use the '_list' meta-rule here too.

```
specification: ‘CILspec’(symbols, slots, axioms, assums,
                rules, goals, envs)

symbols: ‘symbols’(symbol_list)

symbol: ‘symbol’(ident, status, ids, ident, props)

status: ‘type’ | ‘op’ | ‘var’ | ‘pvar’

ids: ‘ids’(ident_list)

props: ‘props’(property_list)

slots: ‘slots’(slot_list)

slot: ‘slot’(term, ident, number)

axioms: ‘axioms’(stmt_list)

stmt: equation | term

equation: ‘eqn’(term, term)

term: ident | fncall

fncall: ident(term_list)

assums: ‘assums’(loc_list)

loc: ‘loc’(nodes, statement)

nodes: ‘nodes’(node_list)
```

```
node: ‘node’(ident, number)

rules: ‘rules’(rule_list)

rule: ‘rule’(facts, ids, facts)

facts: ‘facts’(term_list)

goals: ‘goals’(loc_list)

envs: ‘envs’(env_list)

env: ‘environment’(ident, agents, exposed, order)

agents: ‘agent’(ident, eqns)

eqns: ‘eqns’(equation_list)

exposed: ‘exposed’(terms(term_list))

order: ‘order’(orderspec)

orderspec:
   ident
   | ‘allpar’
   | ‘seq’(orderspec, orderspec)
   | ‘par’(orderspec, orderspec)
```

Appendix B

The Prelude

This appendix specifies the predefined types. The typespecs given here are combined into a file `prelude.cap` that is automatically read and imported by the CAPSL translator prior to user-supplied specifications.

B.1 Basic and Boolean

These types are for basic objects that are not message fields. Note that there is an undeclared "Object" type. No axioms are given for booleans because it is assumed that any protocol analysis tool will have these built in.

```
TYPESPEC BASIC;
TYPES
    Role, Spec, Agent: Object;
    Tspec, Pspec, Espec: Spec;
END;

TYPESPEC BOOLEAN;
IMPORTS BASIC;
TYPES
    Boolean: Object;
CONSTANTS
    true, false: Boolean;
FUNCTIONS
    and(Boolean, Boolean): Boolean, ASSOC, COMM;
    or(Boolean, Boolean): Boolean, ASSOC, COMM;
    not(Boolean): Boolean;
    if(Boolean, Boolean, Boolean): Boolean;
END;
```

B.2 Field

The Field type is the universal supertype for all message fields. There is a subtype Atom of Field for fields that can be detached from the left of a concatenation. A type declaration with no explicit supertype implies a supertype of Atom. A Tape is a nonatomic field; it is a concatenated sequence of atoms. The `ASSOC` property of cat declares that it is associative without the need for explicit axioms.

```
TYPESPEC FIELD;
IMPORTS BOOLEAN;
TYPES
    Field: Object;
    Tape, Atom: Field;
    Principal, Nonce, Number: Atom;
FUNCTIONS
    cat(Field, Field): Tape, ASSOC;
    first(Tape): Atom;
    rest(Tape): Field;
VARIABLES
    X: Atom;
    Y: Field;
AXIOMS
    first(cat(X, Y)) = X;
    rest(cat(X, Y)) = Y;
    INVERT cat(X, Y): X;
    INVERT cat(X, Y): Y | X;
END;
```

B.3 Symmetric-Key Encryption

The basic Skey type is used by several sets of encryption operators. The only operators given in this root typespec are a hash function and a keyed hash (message authentication code). The DES system could be modeled with the `se, sd` pair. The only form of single-operator symmetric system that is commonly seen in practice is `xor`, given below.

```
TYPESPEC SKEY;
IMPORTS FIELD;
TYPES Skey;
FUNCTIONS
    sha(Field): Skey;
    mac(Skey,Field): Skey;
END;

TYPESPEC DSKE;
IMPORTS SKEY;
FUNCTIONS
    se(Skey, Field): Field;
    sd(Skey, Field): Field;
AXIOMS
    se(Skey, Atom): Atom;
    sd(Skey, Atom): Atom;
    sd(K, se(K, D)) = D;
    se(K, sd(K, D)) = D;
    INVERT se(K, D): D | K;
    INVERT sd(K, D): D | K;
END;

TYPESPEC XOR;
IMPORTS SKEY;
FUNCTIONS
    xor(Skey, Skey): Skey, ASSOC, COMM;
AXIOMS
    xor(xor(K,K),K1) = K1;
    INVERT xor(K,K1): K | K1;
    INVERT xor(K,K1): K1 | K;
END;
```

```
/* Symmetric Key Client, Server */

TYPESPEC SKCS;
IMPORTS SKEY;
TYPES Client, Server: Principal;
VARIABLES
        S : Server;
        C : Client;
FUNCTIONS
         csk(Client): Skey, PRIVATE;
         ssk(Server,Client): Skey, PRIVATE;
AXIOMS
         ssk(S, C) = csk(C);
END;

/* Mutual Symmetric Key Node */

TYPESPEC MSKN;
IMPORTS SKEY;
TYPES Node: Principal;
FUNCTIONS
        msk(Node, Node): Skey, COMM, PRIVATE;
END;

/* Arithmetic operations may be used in CAPSL with the
infix syntax +, -, *, /, ^.
*/

TYPESPEC ARITH;
IMPORTS SKEY;
CONSTANTS 1: Skey;
FUNCTIONS
    pls(Skey, Skey): Skey, ASSOC, COMM;
    mns(Skey): Skey;
    tms(Skey, Skey): Skey, ASSOC, COMM;
    div(Skey, Skey): Skey;
    exp(Skey, Skey): Skey;
/*
AXIOMS
```

```
*/
END;
```

B.4 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;
IMPORTS FIELD;
TYPES Pkey;
FUNCTIONS
    keypair(Pkey, Pkey): Boolean, COMM;
END;

TYPESPEC SPKE;
IMPORTS PKEY;
FUNCTIONS
    ped(Pkey, Atom): Atom;
    ped(Pkey, Field): Field;
AXIOMS
    if keypair(K, K1) THEN ped(K1, ped(K, X)) = X ENDIF;
    if keypair(K, K1) THEN INVERT ped(K, X): X | K1 ENDIF;
END;

/* PPK provides simple standard public/private key lookup. */

TYPESPEC PPK;
IMPORTS PKEY;
TYPES PKUser: Principal;
FUNCTIONS
  sk(PKUser): Pkey, PRIVATE;
  pk(PKUser): Pkey;
AXIOMS
  keypair(sk(P),pk(P));
  INVERT ped(sk(P),X): X | pk(P);
  INVERT ped(pk(P),X): X | sk(P);
```

```
END;
```

B.5 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;
FUNCTIONS
    kap(Skey): Pval;
    kas(Pval, Skey): Skey;
AXIOMS
    kas(kap(Ka),Kb) = kas(kap(Kb),Ka);
END;
```

B.6 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(K, K1) THEN verify(K1,seal(K, X), X) ENDIF;
END;
```

B.7 Timestamps

Timestamps are used simply by assuming that each agent that generates or checks a timestamp holds it initially. Equality comparisons can be used to simulate “nearness.” A more advanced version might check ordering.

```
TYPESPEC TIMESTAMP;
TYPES Timestamp;
END;
```

B.8 List

The List type support the non-associative concatenation operator.

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

B.9 End Prelude Marker

This type is placed at the end of the prelude to mark the separation of symbols and axioms in the prelude from those in user-supplied typespecs. This separation is helpful for connectors that provide built-in support for the prelude types.

```
TYPESPEC ENDPRELUDE;
```

```
CONSTANTS endprelude: Boolean;
AXIOMS
    endprelude = true;
END;
```

Appendix C

CAPSL Examples

This appendix contains two relatively complex examples of CAPSL. The SSL example illustrates conditional selection of subprotocols. The SRP example illustrates use of arithmetic. Both utilize customized subtypes of Principal.

C.1 SSL Handshake

The Secure Socket Layer (SSL) Handshake Protocol, version 3, is an Internet Draft that can be found on the Netscape site, `http://home.netscape.com/eng/ssl3`. This CAPSL example is a partial version that expands only one of the cipher spec options, Diffie-Hellman. RSA and Fortezza-DMS are the others. This version also does not perform client authentication. For simplicity we omit the cipher suite and compression method lists.

The CAPSL text illustrates conditional selection of subprotocols and the `DENOTES` section. The `sha` operator is used wherever hashes are called for, and much of the detailed construction of hashes and key material has been simplified. The method for simplifying hashes is to include the contributing data but ignore ordering, constants, and other details that affect the cryptographic strength but not the logical structure of the protocol. The key agreeement operators are used instead of explicit exponentiation in the Diffie-Hellman exchange, so the base and modulus are not mentioned.

```
TYPESPEC SSLS;
TYPES CertServer: PKUser;
```

```
FUNCTIONS
    T(CertServer): Field; /* PK certificate */
VARIABLES
    Sv: CertServer;
CONSTANTS
    CA: PKUser;    /* Certificate Authority */
AXIOMS
    T(Sv) = {Sv,pk(Sv)}sk(CA);
END;

PROTOCOL SSLHandshake;
IMPORTS SSLS;
TYPES CipherSpec;
VARIABLES
    C: PKUser;
    S: CertServer;
    Rc,Rs: Nonce, CRYPTO;       /* Client/ServerHello.random */
    CS: CipherSpec;
    SID: Nonce;                 /* session id */
    PMS: Skey;                  /* pre-master-secret */
    MS: Skey;                   /* master secret */
    PKs: Pkey;                  /* Server public key pk(S) */
CONSTANTS
    DH, RSA, DMS: CipherSpec;
    SSLDH, SSLRSA, SSLDMS: Pspec;
DENOTES
    MS = sha({PMS,Rc,Rs});
ASSUMPTIONS
    HOLDS C: S, CS;
MESSAGES
    ClientHello.       C -> S: C,Rc,CS;
    ServerHello.       S -> C: S,Rs,CS;
    ServerCertificate. S -> C: T(S)%{S,PKs}sk(CA);
    IF      CS = DH  THEN INCLUDE SSLDH;
    ELSE IF CS = RSA THEN INCLUDE SSLRSA;
    ELSE IF CS = DMS THEN INCLUDE SSLDMS;
/*  ELSE INCLUDE SSLERR; */ ENDIF;
    ENDIF; ENDIF;
GOALS
    SECRET MS;
    PRECEDES C: S | MS,Rs,Rc;
END;

PROTOCOL SSLDH;
IMPORTS SSLHandshake;
VARIABLES
    Yc,Ys: Pval;                /* key agreement public values */
    Xc,Xs: Skey, FRESH,CRYPTO;  /* key agreement secret values */
    D: Field;
MESSAGES
    ServerKeyExchange. S -> C: kap(Xs)%Ys,({sha(kap(Xs))}sk(S))%D;
                       {D}PKs = sha(Ys);
/*  SeverHelloDone.    S -> C: { } */
    ClientKeyExchange. C -> S: kap(Xc)%Yc;
                       PMS = kas(Yc,Xs);/
                       PMS = kas(Ys,Xc);
    ClientFinished.    C -> S: sha({MS,C});
    ServerFinished.    S -> C: sha({MS,S});
END;

/* protocols SSLRSA, SSLDMS, and SSLERR would be needed */
```

C.2 Secure Remote Password (SRP) Protocol

SRP is a protocol in the EKE family designed to defeat password guessing, developed at Stanford. There is a web site for it, `http://srp.stanford.edu/srp/`. This CAPSL specification incorporates a few modifications for simplicity:

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

Appendix D

CIL Output Example

This appendix shows the actual CIL output for the NSPK protocol with a sample environment.

D.1 CAPSL Specification for NSPK

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

ENVIRONMENT Test1;
```

```
  IMPORTS NSPK;
  CONSTANTS
    Alice, Bob: PKUser;
    Mallory: PKUser, EXPOSED;
  AGENT A1 HOLDS
    A = Alice;
    B = Bob;
  AGENT B1 HOLDS
    B = Bob;
  EXPOSED
    {Bob}sk(Alice);
END;
```

D.2 CIL Output for NSPK

This is the CIL output from the CAPSL specification above. Note that the prelude was incorporated, resulting in many symbol table entries and axioms.

```
CILspec(
  symbols(
    symbol(Object,type,ids(),Object,props()),
    symbol(BASIC,op,ids(),Tspec,props()),
    symbol(Role,type,ids(),Object,props()),
    symbol(Spec,type,ids(),Object,props()),
    symbol(Agent,type,ids(),Object,props()),
    symbol(Tspec,type,ids(),Spec,props()),
    symbol(Pspec,type,ids(),Spec,props()),
    symbol(Espec,type,ids(),Spec,props()),
    symbol(BOOLEAN,op,ids(),Tspec,props()),
    symbol(Boolean,type,ids(),Object,props()),
    symbol(true,op,ids(),Boolean,props()),
    symbol(false,op,ids(),Boolean,props()),
    symbol(and,op,ids(Boolean,Boolean),Boolean,props()),
    symbol(or,op,ids(Boolean,Boolean),Boolean,props()),
    symbol(not,op,ids(Boolean),Boolean,props()),
    symbol(if,op,ids(Boolean,Boolean,Boolean),Boolean,props()),
    symbol(FIELD,op,ids(),Tspec,props()),
symbol(Field,type,ids(),Object,props()),
symbol(Tape,type,ids(),Field,props()),
symbol(Atom,type,ids(),Field,props()),
symbol(Principal,type,ids(),Atom,props()),
symbol(Nonce,type,ids(),Atom,props()),
symbol(Number,type,ids(),Atom,props()),
symbol(cat,op,ids(Field,Field),Tape,props(ASSOC)),
symbol(first,op,ids(Tape),Atom,props()),
symbol(rest,op,ids(Tape),Field,props()),
symbol(A1,var,ids(),Atom,props()),
symbol(X1,var,ids(),Field,props()),
symbol(SKEY,op,ids(),Tspec,props()),
symbol(Skey,type,ids(),Atom,props()),
symbol(sha,op,ids(Field),Skey,props()),
symbol(mac,op,ids(Skey,Field),Skey,props()),
symbol(DSKE,op,ids(),Tspec,props()),
symbol(se,op,ids(Skey,Atom),Atom,props()),
symbol(sd,op,ids(Skey,Atom),Atom,props()),
symbol(se,op,ids(Skey,Field),Field,props()),
symbol(sd,op,ids(Skey,Field),Field,props()),
symbol(K1,var,ids(),Skey,props()),
symbol(K11,var,ids(),Skey,props()),
symbol(XOR,op,ids(),Tspec,props()),
symbol(xor,op,ids(Skey,Skey),Skey,props(ASSOC,COMM)),
symbol(SKCS,op,ids(),Tspec,props()),
symbol(Client,type,ids(),Principal,props()),
symbol(Server,type,ids(),Principal,props()),
symbol(S1,var,ids(),Server,props()),
symbol(C1,var,ids(),Client,props()),
symbol(csk,op,ids(Client),Skey,props(PRIVATE)),
symbol(ssk,op,ids(Server,Client),Skey,props(PRIVATE)),
symbol(MSKN,op,ids(),Tspec,props()),
symbol(Node,type,ids(),Principal,props()),
symbol(msk,op,ids(Node,Node),Skey,props(COMM,PRIVATE)),
symbol(ARITH,op,ids(),Tspec,props()),
symbol(1,op,ids(),Skey,props()),
symbol(pls,op,ids(Skey,Skey),Skey,props(ASSOC,COMM)),
symbol(mns,op,ids(Skey),Skey,props()),
symbol(tms,op,ids(Skey,Skey),Skey,props(ASSOC,COMM)),
symbol(div,op,ids(Skey,Skey),Skey,props()),
symbol(exp,op,ids(Skey,Skey),Skey,props()),
symbol(PKEY,op,ids(),Tspec,props()),
symbol(Pkey,type,ids(),Atom,props()),
symbol(PK1,var,ids(),Pkey,props()),
symbol(PKI1,var,ids(),Pkey,props()),
symbol(keypair,op,ids(Pkey,Pkey),Boolean,props(COMM)),
symbol(SPKE,op,ids(),Tspec,props()),
symbol(ped,op,ids(Pkey,Atom),Atom,props()),
symbol(ped,op,ids(Pkey,Field),Field,props()),
symbol(PPK,op,ids(),Tspec,props()),
symbol(PKUser,type,ids(),Principal,props()),
symbol(sk,op,ids(PKUser),Pkey,props(PRIVATE)),
symbol(pk,op,ids(PKUser),Pkey,props()),
symbol(PKU1,var,ids(),PKUser,props()),
symbol(KEYAGREEMENT,op,ids(),Tspec,props()),
symbol(Pval,type,ids(),Atom,props()),
symbol(kap,op,ids(Skey),Pval,props()),
symbol(kas,op,ids(Pval,Skey),Skey,props()),
symbol(PKSeal,op,ids(),Tspec,props()),
symbol(Pseal,type,ids(),Atom,props()),
symbol(seal,op,ids(Pkey,Field),Pseal,props()),
symbol(verify,op,ids(Pkey,Pseal,Field),Boolean,props()),
symbol(TIMESTAMP,op,ids(),Tspec,props()),
symbol(Timestamp,type,ids(),Atom,props()),
symbol(LIST,op,ids(),Tspec,props()),
symbol(List,type,ids(),Atom,props()),
symbol(con,op,ids(Field,Field),List,props()),
symbol(head,op,ids(List),Field,props()),
symbol(tail,op,ids(List),Field,props()),
symbol(X1,var,ids(),Field,props()),
symbol(Y1,var,ids(),Field,props()),
symbol(ENDPRELUDE,op,ids(),Tspec,props()),
symbol(endprelude,op,ids(),Boolean,props()),
symbol(NSPK,op,ids(),Pspec,props()),
symbol(A,pvar,ids(),PKUser,props()),
symbol(B,pvar,ids(),PKUser,props()),
symbol(Na,pvar,ids(),Nonce,props(CRYPTO,FRESH)),
symbol(Nb,pvar,ids(),Nonce,props(CRYPTO,FRESH)),
symbol(Test1,op,ids(),Espec,props()),
symbol(Alice,op,ids(),PKUser,props()),
  symbol(Bob,op,ids(),PKUser,props()),
  symbol(Mallory,op,ids(),PKUser,props(EXPOSED)),
  symbol(A1,op,ids(),Agent,props()),
  symbol(B1,op,ids(),Agent,props()),
  symbol(roleA,op,ids(),Role,props()),
  symbol(roleB,op,ids(),Role,props()),
  symbol(UNK,pvar,ids(),Principal,props())
),
slots(
  slot(A,roleA,1),
  slot(B,roleA,2),
  slot(B,roleB,1),
  slot(Na,roleA,3),
  slot(A,roleB,2),
  slot(Na,roleB,3),
  slot(Nb,roleB,4),
  slot(Nb,roleA,4)
),
axioms(
  eqn(first(cat(Al,Xl)),Al),
  eqn(rest(cat(Al,Xl)),Xl),
  invertible(cat(Al,Xl),Al,terms()),
  invertible(cat(Al,Xl),Xl,terms()),
  eqn(sd(Kl,se(Kl,Xl)),Xl),
  eqn(se(Kl,sd(Kl,Xl)),Xl),
  invertible(se(Kl,Xl),Xl,terms(Kl)),
  invertible(sd(Kl,Xl),Xl,terms(Kl)),
  eqn(xor(xor(Kl,Kl),K1l),K1l),
  invertible(xor(Kl,K1l),Kl,terms(K1l)),
  invertible(xor(Kl,K1l),K1l,terms(Kl)),
  eqn(ssk(Sl,Cl),csk(Cl)),
  if(keypair(PKl,PKIl),eqn(ped(PKIl,ped(PKl,Xl)),Xl),true),
  keypair(sk(PKUl),pk(PKUl)),
  invertible(ped(sk(PKUl),Xl),Xl,terms(pk(PKUl))),
  invertible(ped(pk(PKUl),Xl),Xl,terms(sk(PKUl))),
  eqn(kas(kap(Kl),K1l),kas(kap(K1l),Kl)),
  eqn(keypair(PKl,PKIl),verify(PKIl,seal(PKl,Xl),Xl)),
  eqn(head(con(Xl,Yl)),Xl),
  eqn(tail(con(Xl,Yl)),Yl),
  invertible(con(Xl,Yl),Xl,terms()),
  invertible(con(X1,Y1),Y1,terms()),
  eqn(endprelude,true)
),
assums(loc(nodes(node(roleA,0),node(roleB,0)),holds(A,ids(B)))),
rules(
  rule(facts(),ids(),facts(state(roleA,0,terms(A,B)))),
  rule(facts(),ids(),facts(state(roleB,0,terms(B)))),
  rule(
    facts(state(roleA,0,terms(A,B))),
    ids(Na),
    facts(state(roleA,1,terms(A,B,Na)),msg(A,B,terms(ped(pk(B),cat(A,Na)))))
  ),
  rule(
    facts(state(roleB,0,terms(B)),msg(UNK,B,terms(ped(pk(B),cat(A,Na))))),
    ids(Nb),
    facts(
      state(roleB,2,terms(B,A,Na,Nb)),
      msg(B,A,terms(ped(pk(A),cat(Na,Nb))))
    )
  ),
  rule(
    facts(
      state(roleA,1,terms(A,B,Na)),
      msg(UNK,A,terms(ped(pk(A),cat(Na,Nb))))
    ),
    ids(),
    facts(state(roleA,3,terms(A,B,Na,Nb)),msg(A,B,terms(ped(pk(B),Nb))))
  ),
  rule(
    facts(state(roleB,2,terms(B,A,Na,Nb)),msg(UNK,B,terms(ped(pk(B),Nb)))),
    ids(),
    facts(state(roleB,3,terms(B,A,Na,Nb)))
  )
),
goals(
  loc(nodes(node(roleA,3),node(roleB,3)),secret(Na,ids())),
  loc(nodes(node(roleA,3),node(roleB,3)),secret(Nb,ids())),
  loc(nodes(node(roleA,3),node(roleB,3)),precedes(A,B,ids(Na))),
  loc(nodes(node(roleA,3),node(roleB,3)),precedes(B,A,ids(Nb)))
),
  envs(
    environment(
      Test1,
      agents(
        agent(A1,eqns(eqn(A,Alice),eqn(B,Bob))),
        agent(B1,eqns(eqn(B,Bob)))
      ),
      exposed(terms(ped(sk(Alice),Bob))),
      order(allpar)
    )
  )
)
```