A CAPSL Connector to Athena
(Extended Abstract)

Jonathan K. Millen
SRI International
333 Ravenswood Ave
Menlo Park, CA 94025 USA
millen@csl.sri.com

1 Introduction

CAPSL, a Common Authentication Protocol Specification Language, is a high-level language to support security analysis of cryptographic authentication and key distribution protocols. CAPSL specifies a protocol with an abstract form of its message sequence, as is often done in textbooks and articles on authentication protocols. CAPSL and its role in protocol verification is summarized in [3], and project information can be found at the web site `http://www.csl.sri.com/~millen/capsl`.

The central feature of CAPSL support is a translator from CAPSL to an intermediate language called CIL. CIL is based on the multiset rewrite rule (MSR) notation of Cervesato, et al [1]. This notation can be regarded as either an extension of multiset rewriting with a kind of existential quantification, or a Horn fragment of linear logic. It was chosen because it expresses state-transition rules in a pattern-matching style, with symbolic terms to represent encryption and other computations. Representations of this kind are used by many protocol verification and model-checking tools. CIL incorporates these rewrite rules for the procedural part of the specification and also includes symbol-table and other declarative information.

CIL is used by translating it into whatever input format is required by different security analysis tools. Such a translator is called a "connector." It was our hope that connectors would be easy to write. We have had some experience writing connectors for inductive protocol analysis using PVS, model checking using Maude, and model checking using Athena. This note reports on the method for developing connectors from CIL, with emphasis on the Athena connector. This connector, like the others, is still preliminary and experimental, and it is not in "production" use.

2 CAPSL

The salient features of CAPSL are summarized here briefly for background on certain issues that arise, and to motivate the form of CIL specifications. A CAPSL specification is a sequence of modules, each of which is of one of three kinds: a datatype, a protocol, or an environment. For simple protocols, it is possible to get by with a single protocol specification module. Here is an example of a protocol specification that illustrates several CAPSL features; this is the Needham-Schroeder public-key protocol.

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

```
  SECRET Na: A, B;
  SECRET Nb: A, B;
  PRECEDES B: A | Na;
  PRECEDES A: B | Nb;
END;
```

The protocol makes implicit use of certain datatype specifications that define the type PKUser as a subtype of Principal, the key type Pkey for public keys, and the function `pk` that maps from a principal to its public key. The necessary datatype specifications comprise the standard CAPSL "prelude," which is treated as built-in (though it can be changed). Additional datatype specifications can also be supplied.

Protocol variables may have properties, such as "CRYPTO" above for `Na`, conveying something about the choice of values for that variable that is not easily or possibly expressed by refining the data type. Here, "CRYPTO" means that the value is unguessable, so that an attacker should not be able to generate the same value independently.

The "SECRET" and "PRECEDES" goals express confidentiality and agreement policy requirements; they are discussed in more detail in the Goal Generation section.

An environment specification module contains information relevant to setting up particular protocol processes, or "agents," that are to participate in a model checking run. Agents are instances of protocol roles owned by the first-named principal. The role is implied by the variable name for that principal (variable names are imported from the protocol specification). An environment might contain several agents with the same owner, playing the same or different roles. A simple environment might be:

```
ENVIRONMENT Test1;
IMPORTS NSPK;
CONSTANTS
    Alice, Bob: PKUser;
    N1,N2: Nonce;
AGENT Alice1 HOLDS
    A = Alice;
    B = Bob;
    Na = N1;
AGENT Bob1 HOLDS
    B = Bob;
    Nb = N2;
END;
```

3 CIL

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

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

We will argue that all of these components are necessary in the long run for CIL to support translation to analysis tools, but we also find that workable versions of connectors can sometimes skip much of the available information.

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 modules, so there must exist a single global context in which they can all be defined consistently. Program variables must be unique to a protocol specification.

The slot table is a mapping from argument positions in state facts to program variables; this can be explained more easily after state facts are discussed below.

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)`. (Here, `roleA` is a constant of type Role. Role names are formed as *role*+X where X is a variable of type Principal. 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 rules express state transitions using two kinds of MSR "facts:" state facts and message facts. The CAPSL message `A -> B: A,N` generates two rules, one for the sender and one for the receiver. The sender's rule, if it begins in state 0, is (in one version of the MSR notation) $A_0(A,B) \rightarrow (\exists N)A_1(A,B,N), M(A,B,[A,N])$. (This assumes that N is a nonce, and, unlike [1], the message arguments include the sender and receiver, mostly for the sake of supporting translation to Paulson's trace representation [6].) In this kind of rewrite rule, facts mentioned on the left are deleted from the multiset (representing the current global state) and replaced by the facts on the right. Thus, states are updated and received messages are deleted.

The CIL version of this would be

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

Usually, when a principal receives a message, its next transition transmits a reply message. The two consecutive rules can usually be combined into a single rule with the received message on the left and the transmitted message on the right. This rule-merging step is performed in the CAPSL translator by an optimization stage, described in [4].

The MSR representation of a protocol also includes facts and rules associated with intruder activity. These are not included in CIL output, because the intruder model is, for the most part, fixed by the analysis tool. The only per-protocol aspects of the intruder model would be the ability to compute cryptographic and other functions not included in the prelude, plus initial knowledge of some constants assumed to be compromised in a particular scenario.

Now we can explain slots. In a rewrite rule, the variables A, B, etc. are dummy variables, and the rule would have the same meaning if their names were changed to X, Y, etc. However, assumptions and goals refer to program variables, and they must be enforced on the state fact representation.

For example, if we assume that B has the value `Bob` in the initial state of `Alice1`, 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.

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.

4 Example Connector: Athena

Athena is a model checker for security protocols [8], based on the strand space representation [9]. The required input format for Athena was obtained from draft material kindly supplied by 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 [2], 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. Software engineering considerations for expediting connector writing.

These issues are discussed below as they arose in the Athena connector, but analogous issues arose in the PVS and Maude connectors, and similar issues are likely to come up in other connectors as well.

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 [2], 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.

4.2 Normalization: Non-Message Rules

A normalized MSR rule in [2] 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.

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.

4.4 Goal Generation

CAPSL's support for security goals is in flux, but the current basic options are sufficient for Athena. One can state secrecy and agreement goals in CAPSL syntax, and the translator passes them (along with an indication of the final state) to the connector.

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. The `PRECEDES` goal is a form of both Lowe's "agreement" [5] and Schneider's "precedes" property [7].

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.

4.5 Software Engineering Considerations

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.

The Athena-specific parts of the Athena connector consist of less than a page of the user-invoked `cil2athena` class, a six-page workhorse `Athens` class, and a two-page `Strand` class for the strand data structure, built on the tree class. The work took about two weeks calendar time, not full time.

5 Conclusion

The experience of writing an Athena connector had a similar flavor to that of writing an earlier PVS inductive proof connector (not yet documented). The full generality of CAPSL was not exercised, partly because it was not necessary to meet the needs of the analysis tool, and partly because it creates problems in translation that would make the connector much more difficult. Future versions of these connectors and those for other tools might attempt to be more flexible, at least with respect to datatype extensions. The connector support classes developed in Java for parsing CIL and managing an internal tree-structured data representation should be useful to future connector efforts. (We should also acknowledge the convenience of JavaCC for constructing the parser and supplying the basis of the tree class.)

Acknowledgement The author is grateful for the assistance of Dawn Song in providing documentation for the Athena protocol representation language, and for testing the connector.

References

[1] 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.

[2] 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.

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

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

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

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

[7] S. Schneider. Verifying authentication protocols in CSP. *IEEE Transactions on Software Engineering*, 24(9):741–758, September 1998.

[8] 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.

[9] 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.