Design of a CIL Connector to Maude*

Grit Denker
Computer Science Laboratory
SRI International
333 Ravenswood Ave
Menlo Park, CA 94025, USA
denker@csl.sri.com

Abstract

The CAPSL Integrated Protocol Environment effort aims at providing an intuitive and expressive language for specifying cryptographic authentication and key distribution protocols and supporting interfaces to various analysis tools. The CAPSL Intermediate Language CIL has been designed with the emphasis on simplifying translators from CIL to other analysis tools. In this paper we describe the design of a CIL-to-Maude connector. Maude is a rewriting logic based, efficiently executable specification language that has been extended by a model checking tool. We describe how CIL concepts are translated in Maude and propose several optimization techniques in order to achieve protocol specifications that are efficiently executable and analyzable with the Maude model checker. A prototype connector has been implemented in Java.

1 Introduction

Designing cryptographic protocols is difficult. Cryptographic protocols are vulnerable to message modification attacks and it is surprisingly difficult to get even small protocols right. Formal methods have proven helpful for both cryptographic protocol design and analysis. The use of formal languages, including state machines, epistemic logics, and process algebras, supports the rigorous formalization of protocol models and their properties. Moreover, they provide a basis for using tools such as model checkers and theorem-provers to prove protocols correct or uncover security flaws. Unfortunately, most formal analysis tools require protocols to be expressed in a tool-specific language, with special conventions peculiar to cryptographic protocol analysis. The language may be low-level and based on the general-purpose language that the tool was implemented in or designed to support; or it may be a customized higher-level language.

The CAPSL Integrated Protocol Environment [DM00, Mil00] is an effort to provide an intuitive and expressive language for specifying cryptographic authentication and key distribution protocols and to support interfaces to various analysis tools. This way, protocol designer no longer need to repeat the efforts of specifying and preparing a protocol for individual analysis tools.

CAPSL, a Common Authentication Protocol Specification Language, has been designed as a high-level language for security protocols. The core feature is a message list like the ones that are often used to present cryptographic protocols in articles and textbooks. CAPSL is translated to CIL [DM99], a CAPSL Intermediate Language, expressing state transitions with term-rewriting rules. It turns out that CIL is close to the state-transition representation used by almost all of the inductive proof and model-checking tools. The Integrated Protocol Environment provides parser and type checker for CAPSL and CIL and a CAPSL-to-CIL translator.

*Supported by DARPA through Air Force Research Laboratory under Contract F30602-98-C-0258 and by DARPA and NASA through Contract NAS2-98073.

Connectors are being written to adapt CIL to supply input to a variety of formal security analysis tools, including PVS for inductive verification, and Athena and Maude for model-checking. For each connector one has to solve several issues in order to deliver specifications that 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 this paper we describe the design decisions and optimization solutions for a CIL connector to Maude [CELM96]. 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 [Mau99]). Particularly, Maude can be efficiently used as a model checker for security protocols as shown in [DMT98]. [DMT98] illustrated the use of the Maude model checker in finding the well-known attack [Low96] of the NSPK protocol. In this first analysis effort, we spent several weeks on defining a Maude specification for NSPK, appropriately expressing the security goals, and defining the attacker model in Maude. Only the model checking analysis strategy was reusable in other protocol applications. In this paper we describe the design of 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:

- Algorithm for translating protocol specific CIL constructs such as cryptographic operator declarations and message lists into Maude;
- Algorithm for translating environment specific CIL constructs into initial scenarios for the Maude model checker;
- Definition of a general attacker model;
- 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.

2 A CIL Example

We provide a brief description of the CIL concepts by means of the Needham-Schroeder-Public-Key protocol example. The following is the CAPSL specification of NSPK.

```
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 principals `A` and `B` are declared to be of type `PKUser` because principals of that type have the public-key function `pk` defined for them, as well as the corresponding private-key function `sk`. There is an abstract datatype specification defining this subtype in the CAPSL prelude containing invertibility axioms such as `INVERT ped(sk(U),F): F | pk(U)` and `INVERT ped(pk(U),F): F | sk(U)`. The first invertibility axiom states that a field `F` encrypted with the private key `sk(U)` can be decrypted if the public key `pk(U)` is known.

There are three messages in this protocol. In the first message, principal `A` sends the encrypted concatenation of a newly generated nonce `Na` and its address to another principal `B`. `A` uses `B`'s public key for encryption. If the secret key of `B` is not compromised then only `B` can decrypt this message. Principal `B` replies by sending the nonce `Na` and a newly generated nonce `Nb` back to agent `A`. In the third message principal `A` acknowledges the receipt of nonce `Nb`.

There are two types of goals specified: secrecy and agreement goals. Both nonces should remain secret to the agents in roles `A` and `B`. A secrecy goal is violated, if an intruder knows a nonce which is used by an honest agent in a session with another honest agent. The agreement goal uses the keyword `PRECEDES`. `PRECEDES B : A | Na` means that if a `B`-agent reaches its final state, then there exists an `A`-agent that holds the same values as the `B`-agent on the variables `A`, `B`, and `Na`.

CIL [DM99] is a term-rewriting based notation that is capable of expressing all aspects of security protocols in a succinct, precise, and uniform way. The CIL representation of a CAPSL protocol, automatically generated by the CAPSL-to-CIL translator, consists of symbol table, slot table, axioms, localized assumptions, protocol rewrite rules, localized goals, and environment information.

The symbol table of CIL lists the types of variables and the type signatures of functions, with entries of the form `symbol`(*ident, status, args, val, props*) where *ident* is the name of the variable, constant, function, or type, *status* is `var` for dummy variable (from an axiom), `pvar` for a protocol variable, `op` for a constant or function name, or `type` for a type name. The *args* entry is a list of the argument types (empty for a variable, constant, or type), and *val* is the value type or, in case of type specifications, the supertype. Finally, *props* is a list of the property keywords that were given in the original CAPSL declaration (e.g., FRESH).

The symbol table for NSPK includes declarations for cryptographic types, such as nonces (e.g., `symbol(Nonce,type,ids(),Atom,props())`, declarations for operators such as public-key encryption (e.g., `symbol(ped,op,ids(Pkey,Atom),Atom,props())`), definitions of protocol roles (e.g., `symbol(roleA,op,ids(),Role,props())`) and protocol variable declarations (e.g., `symbol(A,pvar,ids(),PKUser,props())` and `symbol(Na,pvar,ids(),Nonce,props(CRYPTO,FRESH))`).

The slot table relates state components with protocol variables. The fact that the value held for variable `A` is the first component in the state of `roleA` (after the required items of the role and state number) is indicated by the slot table entry `slot(A,roleA,1)`. State component positions depend only on the role and variable and not on the state number. The reason that we need slot information is to interpret goals and environment specifications, which mention protocol variables, in terms of the state facts.

Axioms describe algebraic properties of declared operation symbols using equations, boolean terms, if-then-else constructs, or the predefined invertibility predicate (e.g., `invertible(ped(sk(U), F),F,terms(pk(U)))`).

Localized assertions are currently not used for the CIL-to-Maude connector, and, therefore, will not be further discussed.

Local computations of agents and message exchanges between agents are represented by protocol rewrite rules. A rewrite rule describes a state transition of an agent or intruder as a triple: `rule(facts(...),`*identlist*`,facts(...))`, where `facts(...)` are either state-, message-, or attacker-facts, and *identlist* is a list of names of freshly generated variables. In CIL an agent state is represented by a fact `state`(*role, num, termlist*) where role is a constant corresponding to a principal, *num* is a state label, usually an integer, and the terms in the *termlist* are the "memory" of the agent in that role and state. A message is a fact `msg`(*sender, receiver, termlist*). A `net`(*term*) fact represents attacker knowledge. Initialization rules are empty on the lefthand side. E.g., the initialization rule for agent `A` in NSPK is

```
rule(facts(),
     ids(),
     facts(state(roleA,0,terms(A,B)))) .
```

The following rule describes `A`'s state transition in sending the first message of NSPK.

```
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(Na,A))))))
```

Agent `A` starts in a state with `A` and `B` in its memory. In the above state transition, it generates a new nonce `Na`, stores it in the successor state and sends a message with appropriate content to `B`. Semantically, this rewrite rule is enabled in the state of a system in which the lefthand side facts are present, and as an effect of firing the rule, the lefthand side facts are replaced by the righthand side facts of the rule.

The next CIL rule corresponds to the receipt of this message by the agent in role `B`.

```
rule(facts(state(roleB,0,terms(B)),msg(UNK,B,terms(ped(pk(B),cat(Na,A)))),
     ids(),
     facts(state(roleB,1,terms(B,Na,A))))
```

Note that we have adopted the convention where the message source and destination arguments are the true source and intended destination of the message, and they are not accessible neither checked by the receiver of the message. Hence, the free variable `UNK` is used in the message.

Security goals in CAPSL include secrecy and agreement (used for authentication) assertions among others. So far we have only worked out the treatment of secrecy and agreement in the Maude connector.

The current CAPSL-CIL translator passes goals mainly unchanged to CIL, only relating them to the final state of the agents. For instance, `SECRET Na : A, B` and `PRECEDES B : A | Na;` appear in CIL as

```
loc(nodes(node(roleA,3),node(roleB,3)), secret(Na,ids(A,B))),
loc(nodes(node(roleA,3),node(roleB,3)), precedes(B,A,ids(Na))) .
```

An environment specification defines constants of type Principal and uses them to set up protocol sessions, or, more precisely, agents. An agent is an instance of a protocol role for which all initial values are defined. For example, a test run of NSPK with two concurrently existing agents "Alice" and "Bob", in which "Alice" is the initiator is expressed as `environment(Test1, agents(agent(AG1,eqns(eqn(A,Alice),eqn(B,Bob))),agent(AG2,eqns(eqn(B,Bob)))),exposed(),order (AG1 || AG2))`. In the `exposed(...)` part compromised terms such as keys can be declared. The `order(...)` section allows to set up sequential (`;`) or concurrent (||) agents processes in a protocol session.

3 Maude

Maude [CELM96, Mau99] 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 our 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* and *operators*. Operators are introduced with the `op` keyword. They can have user-definable syntax, with underbars '_' marking the argument positions, and are declared with the sorts of their arguments and the sort of their result. 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.

There are 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`.

4 CIL-to-Maude Connector

4.1 Symbols, Axioms, Rules, and Environment

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*. For instance, `sort Nonce` and `subsort Nonce < Atom` are part of the NSPK Maude specification.

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

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 reflect this in the attacker model (see Section 4.3). The rules that enable the attacker to compose and decrypt messages may only involve public functions or functions that are private to the attacker.

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. In appendix A one can see that we enhanced all rewrite rules with such a counter.

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. The 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 Maude as follows

```
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 .
```

The invertibility axiom is translated using the predefined boolean predicate `op INVERT_:_|_ : Field Field Field -> Boolean .` that is part of a Maude prelude in which all CIL specific operators are defined.

Rules

According to the CIL model we define sorts and operations for the multiset of facts.

```
sort Fact Facts Role Agent TermList .        *** sorts for building
                                             *** facts and multiset of facts
subsort Fact < Facts .                       *** a fact is the simplest
                                             *** multiset of facts
```

```
ops roleA roleB : -> Role .                   *** op decl from symbol table

op Msg : Principal Principal Fields -> Fact . *** msg fact
op State : Role MachineInt Fields -> Fact .   *** state fact
op Net : Fields -> Fact .                     *** intruder fact
op Cnt : MachineInt -> Fact .                 *** global counter for fresh vars
```

In fact, we provide a standard Maude specification "cilmodel.maude" that defines all sorts, operations and equations necessary to describe the specifics of a CIL model. We then can mirror the CIL rules in Maude applying only few alterations. One alteration is that Maude rewrite rules need a label. The current automated CIL-to-Maude translator numbers the rules consecutively. The second extension 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. For the purpose of abbreviating protocol presentations, multiset of facts and list of fields are enclosed in square brackets in the CIL-to-Maude connector prototype.

The following rules correspond to the CIL rules for NSPK from Section 2 for sending and receiving message one.

```
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. 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.

```
op Test1 : -> Facts .
eq Test1 = [State(roleA,0,[Alice,Bob]),
            State(roleB,0,[Bob]),
            Net([Spy]),Net([Alice]), Net([Bob])] .
```

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

4.2 Goals

A CIL secrecy goal like `loc(nodes(node(roleA,3),node(roleB,3)), secret(Na,ids(A,B)))` 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). For the purpose of efficient model checking in Maude we chose the following implementation. 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) .
```

4.3 Attacker Model

We implemented a general attacker model in Maude that can be reused for other protocols. The attacker rules include intercepting, overhearing, decomposing, composing, and faking a message. Here is an example of the intercept rule:

```
crl [intercept] :
    [Msg(A,B,[TL]), fs] => [Net([TL]), fs]
    if (not(Net([TL]) in fs) and not(spyOf(A) == true)) .
```

The attacker can intercept messages that are not addressed to him and can learn a list of terms `TL` from a message if he does not yet know them. The condition `not(Net([TL]) in fs` restricts the interception of a message with particular content to one instance. Once an intruder has intercepted a message there is no longer the need to intercept a message with the same content again.

The rule for faking a message is

```
crl [fake] :
    [Net([A]), Net([TL]), fs]
    => [Net([A]), Net([TL]), Msg(Spy,A,[TL]), fs]
    if not(spyOf(A) == true)
       and not(Msg(Spy,a,[TL]) in fs)  .
```

It is sufficient to send the faked messages with `Spy` as sender address, since the CIL rules do not use this address field in order to match the rules.

The decomposition 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 function. In order to achieve determinism for each composing step, the corresponding rule only puts together two fields at a time. The attacker might need to apply this rule several times successively in order to build a composed message which he wants to fake.

Our experimental version is restricted to the operators provided in the CAPSL prelude (i.e., list concatenation, cryptographic operators for symmetric and public key encryption, etc.). Future version will extend this in order to incorporate any user-defined operator for which invertibility axioms are defined.

The NSPK Maude specification produced by the prototype connector is given in Appendix 6.

4.4 Search Strategy and Optimization

Maude's search strategy for model-checking is written on the meta-level of Maude. This allows to import any protocol description and run the search strategy with that protocol.

```
(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. Our depth-first strategy has 4 arguments: the module (protocol), in which the search should take place, an initial test term (as defined by the environment specification), an integer to restrict the depth of search, and a list of rule labels which are to be applied during the search: `op bdfs : Module Term MachineInt QidList -> Strategy .` `Qid` is a Maude specific data type for strings.

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`).

As long as the current depth is smaller than the maximum depth the strategy continues trying to find further rewrites, adding them to the current path. If there are no more rewrites or the maximum depth is reached, the search strategy backtracks. If a step reaches an attack state, the strategy terminates and returns the path to the attack state.

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

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

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

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. The overhear rule of the attacker is an example of such a rule. As long as there are rules for intercepting and faking a message, the effect of an overhear rule can always be simulated by replaying it. We chose to discard the overhear rule in our search strategy. For the Maude model checker it is even more so that discarding the overhear rule speeds up the search time significantly. That is due to the fact that the rule, once applied in a search path, remains applicable in that search path, since the precondition (the message on the lefthand side of the rule) remains in the protocol state. This optimization technique effectively prunes the search tree since some paths are never explored.

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, 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. In [DMKFG00] we proved that such optimizations are attack-preserving. 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.

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 `isInNSPKMsgContentFormat` that is defined to be true for any field that has (partially) the format of a valid NSPK message. The following describes part of the definition of `isInNSPKMsgFormat`.

```
eq Net([A,Na]) isInNSPKMsgContentFormat = true .
eq Net([ped(K,cat(Na,A))]) isInNSPKMsgContentFormat = 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 Conclusion and Future Work

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. There already existed good support for writing connectors in Java. We reused the classes for parsing CIL specifications into labeled trees as well as those classes for manipulating the tree structures. The implementation of the CIL-to-Maude connector took one-person week.

The current connector is based on the predefined data types of CAPSL and supports the optimization strategies discussed in the previous session. Still some restriction apply to this first implementation. User defined data types are not yet handled, neither are order specification in environment declarations. In environment specifications one can specify exposed terms using place holders that have to be expanded to ground facts. One can also define a principal as exposed, meaning that all the secrets of that principal are known to the attacker. These items need 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, and appropriate translations into Maude have to be proposed.

More examples have to be tested and compared with other analysis tools in order to get a feeling for the practicability of our optimization techniques.

Acknowledgements We would like to thank Jon Millen for discussing issues of the CIL connector to Maude, for helping getting the CIL-to-Maude connector started, and for making his Java classes available.

References

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

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

[DM99] G. Denker and J. Millen. CAPSL Intermediate Language. In N. Heintze and E. Clarke, editors, *Workshop on Formal Methods and Security Protocols (FMSP'99), July 5, 1999, Trento, Italy (part of FLOC'99)*, 1999. http://cm.bell-labs.com/cm/cs/who/nch/fmsp99/.

[DM00] G. Denker and J. Millen. CAPSL Integrated Protocol Environment. In D. Maughan, G. Koob, and S. Saydjari, editors, *Proc. DARPA Information Survivability Conference and Exposition, DISCEX2000, January 25-27, Hilton Head Island, SC, USA*, 2000. `http://schafercorp-ballston.com/discex/`.

[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*. IEEE Computer Society, 2000.

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

[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.

[Mau99] Maude Web Site. `http://maude.csl.sri.com/`, 1999.

[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.

[Mil00] J. Millen. CAPSL web site. `http://www.csl.sri.com/~millen/capsl`, 2000.

[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.

6 Generated Maude Specification of NSPK

This Appendix presents the NSPK Maude specification as it is generated by the current CIL-to-Maude connector. The Maude specification of the CAPSL prelude (cryptographic operators and their axiomatization) and the CIL model (multiset of facts, list of fields, etc.) are not presented due to space limitations. The Module ATTACKER imports all those specifications.

```
(mod NSPK is
  protecting ATTACKER .

*** DECLARATIONS

  op NSPK : -> Pspec .
  var A : PKUser .
  var B : PKUser .
  var Na : Nonce .
  var Nb : Nonce .
  op roleA : -> Role .
  op roleB : -> Role .
  var UNK : Principal .
  op Nonce_: MachineInt -> Nonce .
  var fs : Set[Fact] .
  var fl : List[Fields] .
```

```
*** EQUATIONS


*** ENVIRONMENTS

  op Test1 : -> Facts .
  eq Test1 = [
     State(roleA,0,[Alice,Bob]),
     State(roleB,0,[Bob]),
     Net([Spy]),Net([Alice]),Net([Bob])] .


*** RULES

  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] .
  rl [3] :
     [State(roleB,1,[B,Na,A]),
      Cnt(n),fs]
     =>
     [State(roleB,2,[B,Na,A,Nonce (n+1)]),
      Msg(B,A,[ped(pk(A),cat(Na,Nonce (n+1)))]),
      Cnt(n+1),fs] .
  rl [4] :
     [State(roleA,1,[A,B,Na]),
      Msg(UNK,A,[ped(pk(A),cat(Na,Nb))]),
      Cnt(n),fs]
     =>
     [State(roleA,2,[A,B,Na,Nb]),
      Cnt(n),fs] .
  rl [5] :
     [State(roleA,2,[A,B,Na,Nb]),
      Cnt(n),fs]
     =>
     [State(roleA,3,[A,B,Na,Nb]),
      Msg(A,B,[ped(pk(B),Nb)]),
      Cnt(n),fs] .
  rl [6] :
     [State(roleB,2,[B,Na,A,Nb]),
      Msg(UNK,B,[ped(pk(B),Nb)]),
      Cnt(n),fs]
     =>
     [State(roleB,3,[B,Na,A,Nb]),
      Cnt(n),fs] .
```

```
*** ATTACKS

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

endfm)
```