http://www.omnys.it/degas

DEGAS IST-2001-32072

Design Environments for Global ApplicationS

Consortium: UNITN (I) University of Trento (Coordinator), IMM (DK) Informatics and Mathematical Modelling - Technical University of Denmark, DIPISA (I) Dipartimento di Informatica - Università di Pisa, UEDIN (UK) University of Edinburgh - MTCI (I) Motorola Technology Center Italy - OMNYS (I) Omnys Wireless Technology

Deliverable D19

Static Analysers

Contractual date of delivery: December 31, 2003

Actual date of delivery: December 22, 2003 (updated February 10, 2004)

Author(s): Mikael Buchholtz, Hanne Riis Nielson, Flemming Nielson

Participant(s): IMM

Workpackage: WP6

Estimated Person Months: 4.5

Security: Pub **Nature:** P **Version:** 1.1 **Pages:** 15

Abstract.

This document constitutes documentation of a prototype tool for static analysis of security protocols given as processes in the process calculus LySa.

Keyword list: Static analysis, security protocols

Document history

Date	Version	Security	Comments
September 11, 2003	–	Internal	Document start
September 25, 2003	0.1	Public	First draft
December 22, 2003	1.0	Public	Official release
February 10, 2004	1.1	Public	Parser added

Contents

Part I

Executive Summary

This deliverable contains a prototype tool of a static analysis of origin and destination authentication for security protocols. The prototype is downloadable from

http://www.imm.dtu.dk/cs_LySa

This document constitutes the accompanying documentation of the prototype software.

Part II

Full Description

1 Overview

This deliverable contains the implementation of a prototype of the analysis of the process calculus LySa [3, 2] and is implemented in Standard ML of New Jersey (ML) [1]. While this document describes the implementation itself, a description of the technical development behind the implementation can be found in [4]. The implementation is available on the internet as described in Appendix A.

The analysis works on *processes* of the process calculus LySa. The implementation contains three representations of these processes at different levels of abstraction, namely, LySa, Meta LySa, and Labelled LySa. The syntax of the calculus given in [3, 2] corresponds to the representation that is here called LySa. Meta LySa extends this representation with *indexed* constructs that allows for a succinct representation of multiple copies of processes. These indexed constructs are e.g. used in examples in [3] and the idea is that a Meta LySa process can be syntactically expanded into LySa processes with a different numbers of copies of the branches in the indexed constructs. Finally, Labelled LySa is similar LySa but has labels added to the syntax that will be used by the implementation of the analysis as described in [4].

Figure 1 gives an overview of the implementation with the translations between the different representations. The analysis itself takes a Labelled LySa process as input so processes in other representations must be translated before they can be analysed. The analysis produces a formula of Alternation-free Fixed Point logic (ALFP) in ASCII format. This formula serves as the input to the Succinct Solver [5] that solves the formula and returns the analysis result: the smallest analysis components (in their predicate representation [4]) that satisfy the ALFP formula.

$$\begin{tikzcd}
\boxed{\text{Meta LySa}} \arrow[r, bend left, "\text{translate}"] & \boxed{\text{LySa}} \arrow[r, bend left, "\text{translate}"] & \boxed{\text{Labelled LySa}} \arrow[r, bend left, "\text{analyse}"] & \boxed{\text{ALFP}} \arrow[r, bend left, "\text{solve}"] & \boxed{\text{Result}}
\end{tikzcd}$$

Figure 1: Overview of the implementation.

2 Data Structures and Translations

This section contains excerpts from the source files of the implementation. Appendix B contains an overview of these files.

2.1 LySa

In [2] LySa is given according to the following grammar:

$$\begin{array}{rcl}
E & ::= & n \quad | \quad m^+ \quad | \quad m^- \quad | \quad x \quad | \quad \{E_1, \cdots, E_k\}^{\ell}_{E_0}[\mathsf{dest}\ \mathcal{L}] \quad | \\
 & & \{\!|E_1, \cdots, E_k|\!\}^{\ell}_{E_0}[\mathsf{dest}\ \mathcal{L}] \\
P & ::= & \langle E_1, \cdots, E_k\rangle.\,P \quad | \quad (E_1, \cdots, E_j;\ x_{j+1}, \cdots, x_k).\,P \quad | \\
 & & \mathsf{decrypt}\ E\ \mathsf{as}\ \{E_1, \cdots, E_j;\ x_{j+1}, \cdots, x_k\}^{\ell}_{E_0}\ [\mathsf{orig}\ \mathcal{L}]\ \mathsf{in}\ P \quad | \\
 & & \mathsf{decrypt}\ E\ \mathsf{as}\ \{\!|E_1, \cdots, E_j;\ x_{j+1}, \cdots, x_k|\!\}^{\ell}_{E_0}\ [\mathsf{orig}\ \mathcal{L}]\ \mathsf{in}\ P \quad | \\
 & & (\nu\, n)P \quad | \quad (\nu_{\pm}\, n)P \quad | \quad !\,P \quad | \quad P_1|P_2 \quad | \quad \mathsf{0}
\end{array}$$

The grammar is represented by an ML datatype in the ML structure `Lysa` as shown in Figure 2. Names, variables, and crypto-points are all of type `Symbol` and are considered to be the concatenation of the first `string` with all the elements in the `string list`. The annotations are represented by the datatype `Form` where a set of crypto-point is represented as a list. LySa terms, E, are represented by the datatype `Term`: the constructors `NAME`, `NAMEP`, and `NAMEM` represent names n, m^+, and m^-, respectively, while the constructor `ENC` represents symmetric key encryption and the constructor `AENC` represents asymmetric key encryption. Processes, P, are represented by the datatype `Proc`. Note that the sequences of variables in input and decryption are given as a `Term list` though only terms created with the constructor `VAR` may be used here. Similarly, the restricted names in `NEW` and `ANEW` are represented by elements of type `Term` but only terms created with the constructor `NAME` may be used. Note also that parallel composition is given by a list of processes rather than as a binary operator as in the grammar. Finally, the value `CPDY` is the crypto-point $\ell_{\bullet}$ of the attacker and may be used e.g. in protocol specifications.

The structure `Lysa` also contains a auxiliary functions on LySa processes, e.g. to calculate the free names, though these are not shown in Figure 2.

```
structure Lysa =
struct
type Symbol   = string * string list

type Name     = Symbol
type Var      = Symbol
type CP       = Symbol

datatype Form = ORIG   of CP list
              | DEST   of CP list

datatype Term = NAME   of Name
              | NAMEP  of Name
              | NAMEM  of Name
              | VAR    of Var
              | ENC    of Term list * Term * CP * Form option
              | AENC   of Term list * Term * CP * Form option

datatype Proc = OUT    of Term list * Proc
              | INP    of Term list * Term list * Proc
              | DEC    of Term * Term list * Term list * Term *
                          CP * Form option * Proc
              | ADEC   of Term * Term list * Term list * Term *
                          CP * Form option * Proc
              | NEW    of Term * Proc
              | ANEW   of Term * Proc
              | BANG   of Proc
              | PAR    of Proc list
              | NIL

val CPDY      = ...

    ...
end
```

Figure 2: The structure that represents LySA processes (excerpt from the file *lysa.sml*).

```
structure Lysa =
struct
     ...
datatype Proc = ...
              | PAR_I  of string * int * Proc
              | PAR_X  of string * int * string * Proc
              | NEW_I  of ((string * int) list * Term) list * Proc
              | ANEW_I of ((string * int) list * Term) list * Proc
     ...
end
```

Figure 3: The structure that represents Meta LySA processes (excerpt from the file *lysa.sml*).

2.2 Meta LySa

Meta LySa adds *indices* to all names, variables, and crypto-points so they can e.g. be n_i, x_{jk}, or ℓ_{kji}. It also extends the grammar of LySa with indexing constructs so that processes become

$$P \quad ::= \quad \dots \quad | \quad |_{i=a}^{b} \; P \quad | \quad |_{\substack{i=a \\ i \neq j}}^{b} \; P \quad | \quad (\nu_{\bar{i}=\bar{a}}^{b} n_{\bar{i}})P \quad | \quad (\nu_{\pm}{}_{\bar{i}=\bar{a}}^{b} n_{\bar{i}})P$$

The first indexed parallel composition lets the *index* i be used on names, variables, and crypto-points in the process P. The index will take *integer* values from a to b and when the Meta LySa process is translated into a LySa process as described in Section 2.4 it becomes $b - a$ processes in parallel. The second indexed parallel composition is similar except that a process is *not instantiated* when the *value* of index i is equal to the *value* of the index j. Finally, the indexed restriction uses *sequences* of indices, $\bar{i} = i_1 \cdots, i_k$, and corresponding start indices $\bar{a} = a_1, \cdots, a_k$. The construct restricts all the names $n_{i_1 \cdots i_k}$ where each i_j takes values from a_j up to b.

The Meta LySa processes are represented by extending the datatype for LySa processes in the structure `Lysa` as shown in Figure 3. In Meta LySa names, variables, and crypto-points all have the type `Symbol`: the first `string` is the element itself while the `string list` is a list of indices each of which are of type `string`.

The first indexed parallel construct is represented by the constructor `PAR_I`, which contains an index of type `string`, a start index of type `int` and a process. Note that the upper index, i.e. b in the grammar, is not represented in the constructor. The actual value of b will be given when a Meta LySa process is translated to a LySa process as described in Section 2.4. The second indexed parallel composition is represented by constructor `PAR_X` and contains an extra index of type `string` to be excluded in the instantiation. The indexed restrictions are represented by the constructors `NEW_I` and `ANEW_I` that each takes a list of indices of type `string` and corresponding start indices of type `int` together with a name of type `Term` that will be restricted. Note that only terms constructed with `NAME` may be used here.

2.3 Labelled LySa

In [4], Labelled LySa is given according to the following grammar:

$$\begin{array}{lll} E & ::= & n^l \quad | \quad m^{+l} \quad | \quad m^{-l} \quad | \quad x^l \quad | \quad \{E_1, \cdots, E_k\}_{E_0}^{\ell}[\mathsf{dest}\ \mathcal{L}^{l_s}]^l \quad | \\ & & \{\!|E_1^{l_1}, \cdots, E_k^{l_k}|\!\}_{E_0^{l_0}}^{\ell}[\mathsf{dest}\ \mathcal{L}^{l_s}]^l \\ P & ::= & \langle E_1, \cdots, E_k \rangle . \, P \quad | \quad (E_1, \cdots, E_j;\ x_{j+1}, \cdots, x_k). \, P \quad | \\ & & \mathsf{decrypt}\ E\ \mathsf{as}\ \{E_1, \cdots, E_j;\ x_{j+1}, \cdots, x_k\}_{E_0}^{\ell}\ [\mathsf{orig}\ \mathcal{L}^{l_s}]\ \mathsf{in}\ P \quad | \\ & & \mathsf{decrypt}\ E\ \mathsf{as}\ \{\!|E_1, \cdots, E_j;\ x_{j+1}, \cdots, x_k|\!\}_{E_0}^{\ell}\ [\mathsf{orig}\ \mathcal{L}^{l_s}]\ \mathsf{in}\ P \quad | \\ & & (\nu\, n)P \quad | \quad (\nu_{\pm}\, n)P \quad | \quad !\,P \quad | \quad P_1|P_2 \quad | \quad 0 \end{array}$$

Labelled LySa processes are represented by the datatypes in the structure `LLysa` shown in Figure 4. Here names, variables, crypto-points, and *labels* at terms and crypto-points are all of type `string`.

```
structure LLysa =
struct

type Name     = string
type Var      = string
type CP       = string (* Crypto-points *)
type Lab      = string (* Term Labels *)

datatype Term = NAME  of Name * Lab
              | NAMEP of Name * Lab
              | NAMEM of Name * Lab
              | VAR   of Var  * Lab
              | ENC   of Term list * Term * CP * CP list * Lab * Lab
              | AENC  of Term list * Term * CP * CP list * Lab * Lab

datatype Proc = OUT   of Term list * Proc
              | INP   of Term list * Term list * Proc
              | DEC   of Term *  Term list * Term list * Term *
                         CP * CP list * Lab * Proc
              | ADEC  of Term *  Term list * Term list * Term *
                         CP * CP list * Lab * Proc
              | NEW   of Term * Proc
              | ANEW  of Term * Proc
              | BANG  of Proc
              | PAR   of Proc * Proc
              | NIL

val CPDY      = ...

    ...
end
```

Figure 4: The structure that represents Labelled LySA processes (excerpt from the file *llysa.sml*).

Encrypted terms are represented by the constructor `ENC` where the first of the two labels of type `Lab` is the label, l_s, at the set of crypto-points while the second label is the label, l, of the term itself. In addition to what is shown in Figure 4 the structure `LLysa` contains auxiliary functions on Labelled LySA processes.

2.4 Translations

The translation from Meta LySA to LySA is preformed by the function `ml2lProc` in the structure `MLysa2Lysa` as shown in Figure 5. The function returns a LySA process that is the Meta LySA process given as the second argument where all indexed constructs have been unfolded the number of times given by the first argument.

Important: note that when a process is instantiated with index 0 it is implicitly assumed to be a process that communicates with the attacker. Therefore, $\ell_\bullet$ is added to all sets of crypto-points in destination and origin annotations in the 0^{th} unfolding.

```
structure MLysa2Lysa :
  sig
    val ml2lProc : int -> Lysa.Proc -> Lysa.Proc
    ...
  end
```

Figure 5: Structures for translation from Meta LySA to LySA in the file *mlysa2lysa.sml*.

The translation of a LySA process into a Labelled LySA process is performed by the function `l2llProc` in the structure `Lysa2LLysa` as shown in Figure 6. The translation adds *unique* labels to all terms and all sets of crypto-points in the process. This is done through a label counter that may be reset by calling the function `reset`. Annotations of destination and origin at encryption and decryption, respectively, may be ignored by writing `NONE` as annotation. The function `l2llProc` translates this into the equivalent of having no annotation, which is to allow all the crypto-points in the processes being translated as well as $\ell_\bullet$ into the sets of intended destinations or origins. The structure `Lysa2LLysa` additionally contains functions to translate other syntactic categories such as terms and annotations, which are not shown in Figure 6.

Important: note that `l2llProc` removes all characters that *are not* alphanumeric characters or a prime (`'`) in the strings of names, variables, and crypto-points. This may cause name-clashes that result in unexpected analysis results! Furthermore, all the strings in indices should represent integer numbers where negative numbers are prefixed with `-`.

```
structure Lysa2LLysa :
  sig
    val l2llProc : Lysa.Proc -> LLysa.Proc
    val reset : unit -> unit
    ...
  end
```

Figure 6: Structure for translating from LySA to Labelled LySA in the file *lysa2llysa.sml*.

3 The Analysis

The implementation of analysis itself may be found in the file *analysis1.sml* and closely follows [4, Section 6]. As shown in Figure 7, the file contains a structure `Analysis1`. The structure contains a number of values that represent various elements of the attacker which may be useful when inspecting the result. Here, `LDY` represents $l_\bullet$, `LSDY` represents $l_\mathcal{C}$, `NDY` represents $n_\bullet$, `NPDY` represents $m_\bullet^+$, `NMDY` represents $m_\bullet^-$, and `ZDY` represents $z_\bullet$.

The structure contains generation functions `genProc` and `genTerm` corresponding to the function $\mathcal{G}$ in [4] for processes and terms. Furthermore, the function `genNEI` generates a predicate for non-empty intersection of sets of values found when analysing a process while `genDY` generates a suitable attacker for a given process.

```
structure Analysis1 :
sig
    val LDY : string
    val LSDY : string
    val NDY : LLysa.Term
    val NMDY : LLysa.Term
    val NPDY : LLysa.Term
    val ZDY : LLysa.Term

    val canon : LLysa.Term -> LLysa.Name
    val genTerm : LLysa.Term -> string
    val genProc : LLysa.Proc -> string
    val genDY : LLysa.Proc -> string
    val genNEI : LLysa.Proc -> string

    val analyse : LLysa.Proc -> string

  end
```

Figure 7: Structure for the analysis in the file *analysis1.sml*.

Finally, the function `analyse` binds it all together by taking a Labelled LySa process and return an ASCII version of an ALFP formula corresponding to the analysis of that process. In the ASCII version of the formula, the names of the predicates in [4] are changed into ASCII format as summarised in Appendix C.

Important: note that all strings in names, variables, and crypto-points in the Labelled LySa process that is analyses should only contain alphanumeric characters, prime (`'`), or underscore (`_`). Furthermore, none of these strings may start with an underscore. These requirements will automatically be fulfilled if the Labelled LySa process is obtained from a LySa process by calling the function `l2llProc` as described in Section 2.4.

3.1 An Example

Appendix D contains a listing of the file *example.sml* that shows an example of the use of the analysis. It follows closely the procedure sketched in Figure 1.

First, a Meta LySa process `MLP` is declared using the datatype from the structure `Lysa`, which corresponds to the following process

$$(\nu_{ij=11}^{b} K_{ij})\ |_{i=1}^{b}\ |_{\substack{j=1\\ j\neq i}}^{b} \quad \begin{array}{l} \ \ !\,(\nu\, m_{ij})\langle\{m_{ij}\}_{K_{ij}}^{a_{ij}}[\mathsf{dest}\ \{b_{ij}\}]\rangle.\,0 \\ |\ \ !\,(;\ x_{ij}).\ \mathsf{decrypt}\ x_{ij}\ \mathsf{as}\ \{;\ y_{ij}\}_{K_{ij}}^{b_{ij}}\ [\mathsf{orig}\ \{a_{ij}\}]\ \mathsf{in}\ 0 \end{array}$$

that repeatedly sends and receives $b \cdot b$ messages, m_{ij} for $1 \leq i, j \leq b$ and $i \neq j$, encrypted under the respective keys K_{ij} on the network. Next, the Meta LySa process

MLP is translated into a LySa process LP by calling MLysa2Lysa.ml2lProc. This process is then translated into the Labelled LySa process LLP using the function Lysa2LLysa.l2llProc. The Labelled LySa process is given as input to the function Analysis1.analyse that in return produces a string ALFP that contains an ASCII version of the ALFP formula describing the analysis.

Finally, the Succinct Solver is initialised by the call FormulaAnalyzer.init(). The ALFP formula is solved using the function FormulaAnalyzer.solveStd that returns the output, i.e. the analysis components, on standard out. For more on solving functions and handling output from the Succinct Solver see [6].

References

[1] Website for Standard ML of New Jersey. http://www.smlnj.org, 2003.

[2] C. Bodei, M. Buchholtz, P. Degano, F. Nielson, and H. Riis Nielson. Static validation of security protocols. Submitted to Journal of Computer Security 2003.

[3] C. Bodei, M. Buchholtz, P. Degano, F. Nielson, and H. Riis Nielson. Automatic validation of protocol narration. In *Proceedings of the 16th Computer Security Foundations Workshop (CSFW 2003)*, pages 126–140. IEEE Computer Society Press, 2003.

[4] M. Buchholtz. Implementing control flow analysis for security protocols. DEGAS Report WP6-IMM-I00-Pub-003, To appear.

[5] F. Nielson, H. Riis Nielson, and H. Seidl. A succinct solver for ALFP. *Nordic Journal of Computing*, 9:335–372, 2002.

[6] H. Sun. User's guide for the Succinct Solver (V2.0) . http://www.imm.dtu.dk/cs_SuccinctSolver/userGuide.pdf, 2003.

A Installation Guide

The LySa analysis requires the following software to be installed:

- New Jersey SML version 110.0.7 available at

  http:://www.smlnj.org

- The Succinct Solver version 2.0 (release October 17, 2003) available at

  http://www.imm.dtu.dk/cs_SuccinctSolver/

Follow the installation procedures at the respective websites. Next, the LySa analysis itself should be installed:

- Download the file:

  http://www.imm.dtu.dk/cs_LySa/lysatool-1.1.tar.gz

- Unpack the *lysatool-1.1.tar.gz*, which creates a subdirectory called *lysatool*.

- Edit the file *lysatool/sources.cm* and set the path *.../Formulas* to point to the directory *Formulas* in your installation of the Succinct Solver (typically the directory *HORN/Formulas* in your Succinct Solver installation.)

Now, the analysis is ready to run. For example, to run the file *example.sml* found in the directory *lysatool* (1) change to the directory *lysatool*, (2) start ML (e.g. by typing `sml` in a shell), and (3) execute the command `use "example.sml";`. The Succinct Solver will now compute the analysis result for the example process. Note, for example, that the component `PSI` will be empty as expected since all messages are encrypted under keys unknown to the attacker.

B Overview of Source Files

The LySa analysis contains the following files (in the directory *lysatool* described in the installation guide A):

lysa.sml	The definition of structure `Lysa` with the datatype for LySa and Meta LySa and auxiliary functions for these processes.
llysa.sml	The definition of structure `LLysa` with the datatype Labelled LySa and auxiliary functions on Labelled LySa processes.
mlysa2lysa.sml	The definition of the structure `MLysa2Lysa` with functions that translate Meta LySa to LySa.
lysa2llysa.sml	The definition of the structure `Lysa2LLysa` with functions that translate LySa to Labelled LySa.
analysis1.sml	Functions for the origin and destination authentication analysis.
COPYRIGHT	Copyright statement.
set.sml	An implementation of sets and set operations.
sources.cm	Source file for Standard ML of New Jersey's Compilation Manager.
version	Description of version and log of changes.
example.sml	Example program listed in Appendix D.
example2.sml	Example of parsing ASCII input as described in Appendix E.1.

io/	Directory containing auxiliary functions for input/output.
protocols/	Directory containing example protocols.

C ASCII Names of Predicates

The implementation closely corresponds to the description in [4] but uses the following ASCII names for the predicates described in [4]:

Predicate	ACSII Name
ρ_k^g	RHOk
κ_k^p	KAPPAk
ψ	PSI
γ^g	C
δ^g	D

Predicate	ACSII Name
α_k	ALPHAk
σ_k	SIGMAk
NEI	NEI
$\mathcal{N}_{\mathsf{C}}$	N
PQD	PQD

D Listing of the file *example.sml*

```
(*****************************************************************)
(* Example of how the analysis can be used (see D19 Section 3.1)   *)
(*****************************************************************)

CM.make();

let
    local
        open Lysa
    in
    val MLP = NEW_I([([("i",1),("j",1)],NAME("K",["i","j"]))],
              PAR_I("i",1,
              PAR_X("j",1,"i",
              PAR(
                  [BANG(
                  NEW(NAME("m",["i","j"]),
                  OUT([ENC([NAME("m",["i","j"])],
                           NAME("K",["i","j"]),
                           ("a",["i","j"]),
                           SOME(DEST([("b",["i","j"])])))],
                  NIL))),
                  BANG(
                  INP([],[VAR("x",["i","j"])],
                  DEC(VAR("x",["i","j"]),
                      [],
                      [VAR("y",["i","j"])],
                      NAME("K",["i","j"]),
                      ("b",["i","j"]),
                      SOME(ORIG([("a",["i","j"])])),
                  NIL)))])))))
    end

    val LP   = MLysa2Lysa.ml2lProc 3 MLP
    val LLP  = Lysa2LLysa.l2llProc LP

    val ALFP = Analysis1.analyse LLP

    val ss   = FormulaAnalyzer.init();
in
    FormulaAnalyzer.solveStd(ss,FormulaAnalyzer.inputStr(ALFP))
end
```

E Input and Output Functionality

This appendix contains information on auxiliary input and output capabilities for the analysis prototype tool.

E.1 Parsing LySa from ASCII text

Meta LySa processes (including LySa processes) can be parsed from ASCII text strings according to the grammar in Figure 8. Here the set *identifier* contains strings that begin with a letter and are followed by zero, one, or more letters or digits. The set *number* contains strings with decimal representation of numbers using the prefix `-` to denote a negative number. The following list of strings are considered to be *keywords* that cannot be used for other purposed than described by the grammar:

`as`, `at`, `CPDY`, `dest`, `decrypt`, `in`, `orig`, `new`

Any string between an opening `/*` until the first closing `*/` will be regarded as comments and disregarded when parsing.

In the grammar a non-terminal post-fixed with $^+$ denotes non-empty comma-separated list of that non-terminal while a non-terminal post-fixed with * denotes a (possibly empty) comma-separated list of that non-terminal. An ε in the body of a rule denotes the empty string.

The prefix operators on processes (input, output, decryption, and restriction) binds tighter than the `!` at replication that again binds tighter than parallel composition. Parallel composition is left associative. Alternative precedence may be forced by adding parentheses around processes and terms.

Names and variables are parsed according the same syntax and conflicts are resolved by ensuring that any occurrence of a variable that is in scope (of an input or a decryption) will indeed be interpreted as a variable. At all other places elements will

```
proc         :: =  ( proc )  |  < term* > . proc  |  ( term* ; var* ) . proc  |
                   decrypt term as { term* ; var* } : term orig in proc  |
                   decrypt term as {| term* ; var* |} : term orig in proc  |
                   ( new name ) proc  |  ( new +- name ) proc  |
                   ! proc  |  proc | proc  |  0  |
                   |_{ assign } proc  |  |_{ assign \ index } proc  |
                   ( new_{ assign+ } name ) proc  |  ( new_{ assign+ } +- name ) proc

term         ::=   ( term )  |  { term* } : term dest  |  {| term* |} : term dest  |
                   name  |  namep  |  namem  |  var

name         ::=   indentifier subscript

namep        ::=   indentifier + subscript

namem        ::=   indentifier - subscript

var          ::=   indentifier subscript

subscript    ::=   _{ index* }  |  ε

index        ::=   identifier  |  number

assign       ::=   index = number

orig         ::=   [ at cryptopoint orig { cryptopoint* } ]  |  [ at cryptopoint ]  |  ε

dest         ::=   [ at cryptopoint dest { cryptopoint* } ]  |  [ at cryptopoint ]  |  ε

cryptopoint  ::=   identifier subscript  |  CPDY
```

Figure 8: ASCII grammar for LySa.

be interpreted as names. Thus, processes may never contain free variables though they may contain free names.

Annotations of origin and destination information are added in square brackets. The full annotations give both a crypto-point denoting the place of decryption/encryption as well as a set of crypto-points for expected origin/destination. The latter information may be left out and this will be interpreted as the semantic equivalent of having no requirement of origin/destination. If an annotation is altogether empty the decryption/encryption will further more be interpreted as if it takes place at an unspecified crypto-point.

Using the parser. The parser may be accessed through the signature `LysaASCIIIO` shown in Figure 9. Processes may be parsed from a file, a stream, or a string using the functions `parseFile` (the argument is the filename), `parseStream`, and `parseString`, respectively. On unsuccessful parsing the exception `parseError` is thrown.

An example of an ML program that uses the parser and the analysis may be found in the file *example2.sml*. The program uses the LySa file *example.lysa* that may be found in the *protocols* directory and is listed below:

```
structure LysaASCIIIO :
  sig
    exception parseError of string
    val parseFile : string -> Lysa.Proc
    val parseStream : TextIO.instream -> Lysa.Proc
    val parseString : string -> Lysa.Proc
    val toString : Lysa.Proc -> string
    val toFile : string -> Lysa.Proc -> unit
  end
```

Figure 9: Structure for parsing ASCII text into Meta LySa in the file *io/lysaacsiiio.sml*.

```
/* Example of a LySa process in ASCII text format */

(new_{i=1,j=1} K_{i,j})
(
|_{i=1} |_{j=1\i} (
    ! (new m_{i,j}) < { m_{i,j} } : K_{i,j} [at a_{i,j} dest {b_{i,j}}]>.0
    |
    ! (; x_{i,j}).
      decrypt x_{i,j} as {; y_{i,j} } : K_{i,j} [at b_{i,j} orig {a_{i,j}}] in 0
    )
)
```