ELAN

A rule-based programming language

Hélène Kirchner and Pierre-Etienne Moreau
LORIA – CNRS – INRIA
Nancy, France

Introduction

Overview

- General presentation
- ELAN: simple part (computation)
- ELAN: advanced part (deduction)
- Comparison with other formalisms
- Some applications

General presentation

What is ELAN?

ELAN = computation rules + deduction rules

i.e:

ELAN = unlabelled rules + labelled rules + strategies

ELAN's general features

- ELAN has been designed to prototype, experiment and study deduction systems, in particular for constraint solving and theorem proving
- ELAN is a declarative rule-based language
- ELAN has three main originalities w.r.t. other algebraic languages:

ELAN's general features

- ELAN has been designed to prototype, experiment and study deduction systems, in particular for constraint solving and theorem proving
- ELAN is a declarative rule-based language
- ELAN has three main originalities w.r.t. other algebraic languages:
  - associative-commutative operators

ELAN's general features

- ELAN has been designed to prototype, experiment and study deduction systems, in particular for constraint solving and theorem proving
- ELAN is a declarative rule-based language
- ELAN has three main originalities w.r.t. other algebraic languages:
  - associative-commutative operators
  - non-deterministic computations

ELAN's general features

- ELAN has been designed to prototype, experiment and study deduction systems, in particular for constraint solving and theorem proving

- ELAN is a declarative rule-based language

- ELAN has three main originalities w.r.t. other algebraic languages:
  - associative-commutative operators
  - non-deterministic computations
  - a user defined strategy language to control rewriting

Deductions and computations

- Rules for computations:
  - unique normal form required
  - leftmost innermost deterministic strategy fixed
- Rules for deductions:
  - no confluence nor termination required
  - application strategy required
    * non-determinism handled in practice by backtracking
    * strategies are used to express choices
    * strategy may fail

Simple introduction

A very simple example

The Fibonacci function can be specified as follow:

$$\begin{array}{lcl} fib(0) & \rightarrow & 0 \\ fib(1) & \rightarrow & 1 \\ fib(n) & \rightarrow & fib(n-1) + fib(n-2) \text{ if } n > 1 \end{array}$$

In ELAN, it can be written as follow:

```
rules for int
  n : int;
global
  [] fib(0) => 0 end
  [] fib(1) => 1 end
  [] fib(n) => fib(n - 1) + fib(n - 2) if n > 1 end
end
```

Writing a rule

- the left-hand side is a term: we need a syntax
- in ELAN, the notation can be mix-fixed
- we have to give a multi-sorted signature

```
operators global
  fib(@)  : (int) int;
end
```

- `@` stands for a placeholder
- `int` is a sort
- `fib(@)` is a function symbol

Writing a signature

- a sort can be defined

```
sort Natural;
```

- but, we can also import an existing module

```
module fib
import global int;
end
```

A stand-alone Finonacci specification

```
module fib
import global int;
end

operators global
  fib(@)  : (int) int;
end

rules for int
  n : int;
global
   [] fib(0) => 0 end
   [] fib(1) => 1 end
   [] fib(n) => fib(n - 1) + fib(n - 2) if n > 1 end
end
```

Exercise

How to specify the Peano arithmetic?

$$\begin{array}{lcl} x + zero & \rightarrow & x \\ x + suc(y) & \rightarrow & suc(x + y) \end{array}$$

Solution

```
module Peano
sort Nat;
end

operators global
  zero   :            Nat;
  suc(@) : (Nat)      Nat;
  @ + @  : (Nat Nat) Nat;
end

rules for Nat
  x,y : Nat;
global
  [] x + zero   => x        end
  [] x + suc(y) => suc(x+y) end
end
```

Signature

- a function symbol can be associative-commutative

$$f(x, y) = f(y, x) \text{ and } f(x, f(y, z)) = f(f(x, y), z)$$

In ELAN:

```
operators global
  f(@,@)  : (Nat Nat) Nat (AC);
end
```

Example

```
module Peano
sort Nat;
end

operators global
  zero    :            Nat;
  suc(@)  : (Nat)      Nat;
  @ + @   : (Nat Nat) Nat (AC);
end

rules for Nat
  x,y : Nat;
global
   [] x + zero          => x              end
   [] suc(x) + suc(y) => suc(suc(x+y)) end
end
```

Exercise

How to specify the Set data structure?

Examples of normal forms

$$
\begin{array}{ccc}
\emptyset & \leadsto & \emptyset \\
\emptyset \cup a \cup b & \leadsto & a \cup b \\
a \cup a \cup b \cup b & \leadsto & a \cup b \\
a \cup b \cup b & \leadsto & a \cup b \\
& \ldots &
\end{array}
$$

Solution

```
module Set
sort Set, Element;
end

operators global
  empty :           Set;
  @     : (Element) Set;
  @ U @ : (Set Set) Set (AC);
end

rules for Set
  s : Set;
global
  [] s U empty => s end
  [] s U s     => s end
end
```

Is it correct?

$$\begin{array}{rcl} s \cup \emptyset & \longrightarrow & s \\ s \cup s & \longrightarrow & s \end{array}$$

- $\emptyset \rightsquigarrow$

Is it correct?

$$\begin{array}{rcl} s \cup \emptyset & \rightarrow & s \\ s \cup s & \rightarrow & s \end{array}$$

- $\emptyset \rightsquigarrow \emptyset$, because no rule can be applied
- $\emptyset \cup a \cup b \rightsquigarrow$

Is it correct?

$$\begin{array}{rcl} s \cup \emptyset & \longrightarrow & s \\ s \cup s & \longrightarrow & s \end{array}$$

- $\emptyset \rightsquigarrow \emptyset$, because no rule can be applied
- $\emptyset \cup a \cup b \rightsquigarrow a \cup b$, by applying the first rule
- $a \cup a \cup b \cup b \rightsquigarrow$

Is it correct?

$$\begin{array}{rcl} s \cup \emptyset & \rightarrow & s \\ s \cup s & \rightarrow & s \end{array}$$

- $\emptyset \rightsquigarrow \emptyset$, because no rule can be applied
- $\emptyset \cup a \cup b \rightsquigarrow a \cup b$, by applying the first rule
- $a \cup a \cup b \cup b \rightsquigarrow a \cup b$, by applying the second rule
- $a \cup b \cup b \rightsquigarrow$

Is it correct?

$$\begin{array}{rcl} s \cup \emptyset & \longrightarrow & s \\ s \cup s & \longrightarrow & s \end{array}$$

- $\emptyset \leadsto \emptyset$, because no rule can be applied
- $\emptyset \cup a \cup b \leadsto a \cup b$, by applying the first rule
- $a \cup a \cup b \cup b \leadsto a \cup b$, by applying the second rule
- $a \cup b \cup b \leadsto a \cup b$, why?

Is it correct?

$$
\begin{array}{lcl}
s \cup \emptyset & \longrightarrow & s \\
s \cup s & \longrightarrow & s
\end{array}
$$

- $\emptyset \leadsto \emptyset$, because no rule can be applied
- $\emptyset \cup a \cup b \leadsto a \cup b$, by applying the first rule
- $a \cup a \cup b \cup b \leadsto a \cup b$, by applying the second rule
- $a \cup b \cup b \leadsto a \cup b$, why? by applying the second rule on the subterm $b \cup b$

Confluence and Terminaison

- the rewrite system is supposed to be confluent and terminating

```
[] fact(0) => 1          end
[] fact(n) => n*fact(n-1) end
```

- the system is not confluent and does not terminate

`fact(0)` $\leadsto$ `1`

`fact(0)` $\leadsto$ `0*fact(0-1)` $\leadsto$? (`0` or `0*-1*fact(-1-1)`?)

- solution:

Confluence and Terminaison

- the rewrite system is supposed to be confluent and terminating

```
[] fact(0) => 1            end
[] fact(n) => n*fact(n-1) end
```

- the system is not confluent and does not terminate

fact(0) $\rightsquigarrow$ 1
fact(0) $\rightsquigarrow$ 0*fact(0-1) $\rightsquigarrow$? (0 or 0*-1*fact(-1-1)?)

- solution: add a condition

Confluence and Terminaison

- the rewrite system is supposed to be confluent and terminating

```
[] fact(0) => 1            end
[] fact(n) => n*fact(n-1) end
```

- the system is not confluent and does not terminate

fact(0) ⇝ 1
fact(0) ⇝ 0*fact(0-1) ⇝ ? (0 or 0*-1*fact(-1-1)?)

- solution: add a condition

```
[] fact(0) => 1                     end
[] fact(n) => n*fact(n-1) if n>0 end
```

Conditional rewriting

- a rule is applied only if
  - the left-hand side matches the suject
  - the condition is satisfied (has `true` as normal form)

Question

What happens is the associative-conditional case?

consider the rule:

$$x \cup y \cup z \rightarrow z \text{ if } x - y = 2$$

and the suject:

$$1 \cup 2 \cup 3$$

Can we apply the rule?

Answer

$$x \cup y \cup z \to z \text{ if } x - y = 2$$

- yes
- because the rule is tried with all possible substitutions
  - $\sigma_1 = \{x \mapsto 1, y \mapsto 2, z \mapsto 3\}$
  - $\sigma_2 = \{x \mapsto 2, y \mapsto 1, z \mapsto 3\}$
  - $\sigma_3 = \{x \mapsto 1, y \mapsto 3, z \mapsto 2\}$
  - $\ldots$
  - $\sigma_6 = \{x \mapsto 3, y \mapsto 1, z \mapsto 2\}$
- $\sigma_6(x - y = 2) \rightsquigarrow \top$, the rule can be applied
- $1 \cup 2 \cup 3 \rightsquigarrow 2$

ELAN: an Algebraic Specification Formalism

- it is also a programming language (used by humans)
- ELAN is a modular specification system
- a module can be parametrized
- there is also a preprocessor that performs textual replacement

Simple module

- definition of sorts

```
module Peano
sort Nat;  end
```

- definition of operators

```
operators global
  zero     :           Nat;
  suc(@)   : (Nat)     Nat;
  @ + @    : (Nat Nat) Nat;
  @ * @    : (Nat Nat) Nat;
  (@ * @)  : (Nat Nat) Nat alias @+@:;
```

Simple module (cont.)

- definition of rules

```
rules for Nat
  x,y : Nat;
global
  [] x + zero   => x          end
  [] x + suc(y) => suc(x+y)   end
  [] x * zero   => zero       end
  [] x * suc(y) => x + (x*y)  end
end
```

Parametrized module

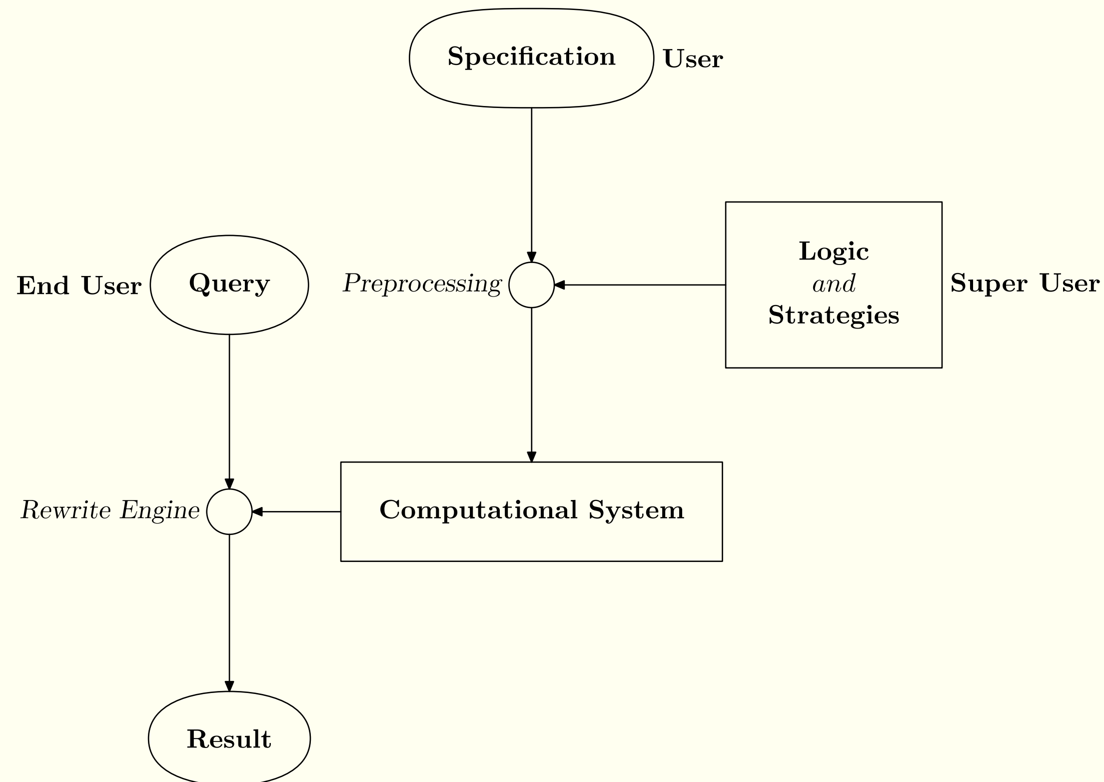
```
module list[X]
  import int;      end
  sort X list[X]; end
  operators global
    nil_X        :                        list[X];
    nil          :                        list[X] alias nil_X:;
    @ . @        : ( X list[X] )          list[X];
    append(@,@)  : ( list[X] list[X] )    list[X];
  end
  rules for list[X]
    e     : X;
    l1,l2 : list[X];
  global
    [] append(nil,l2)   => l2              end
    [] append(e.l1,l2)  => e.append(l1,l2) end
  end
```

Preprocessor

- performs textual replacement
- can generate signatures
- can generate rules
- can generate anything

```
module list[X]
  ...
  operators global
    nil_X : list[X];
    nil   : list[X] alias nil_X:;
  end
```

System organization



Example: Poly

```
rules for poly
  p1, p2 : poly;
  x, y   : poly_variable;
global
   []  deriv(p1+p2,x)     => deriv(p1,x) + deriv(p2,x)  end
   []  deriv(y,x)         => 0    if x!=y               end
   []  deriv(x,x)         => 1                          end
end
```

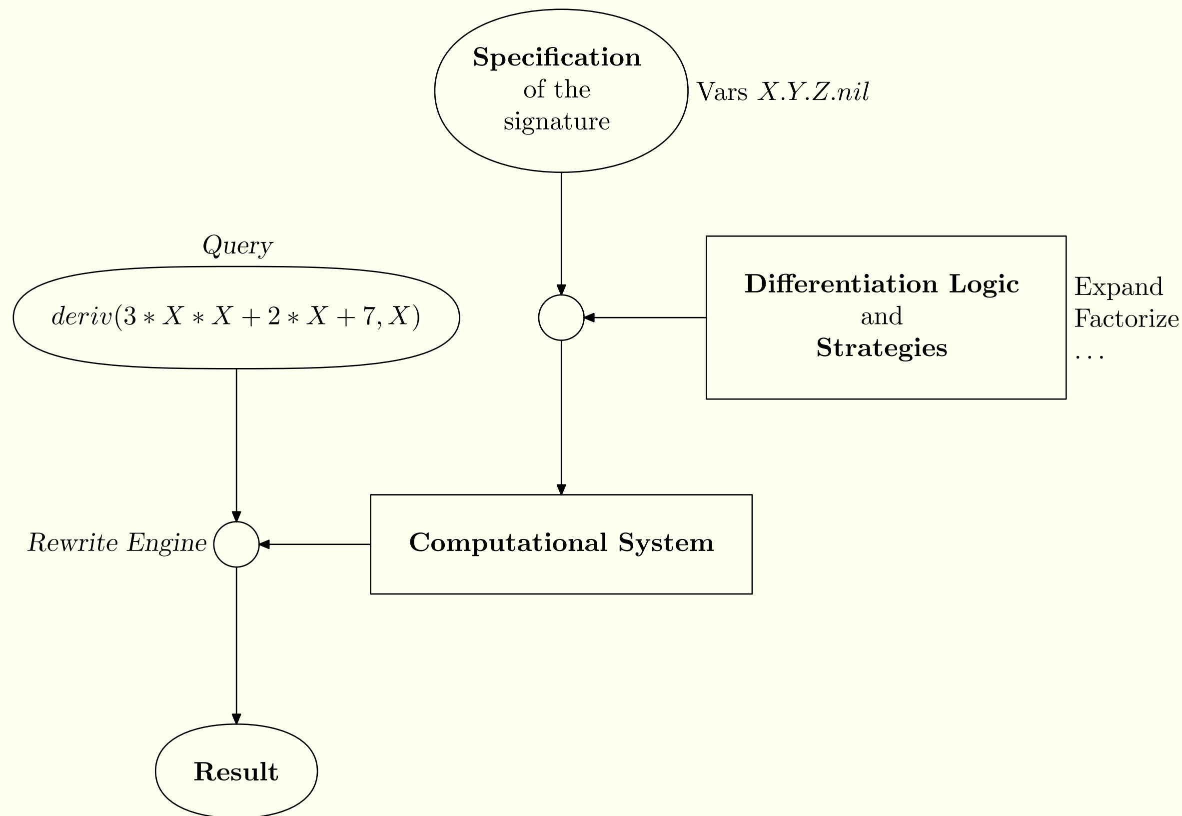
- the set of constants of sort poly_variable should be parametrized at compile time

Definition of poly_variable

```
module poly[Vars]
import global Vars identifier list[identifier]; end
sort poly poly_variable;                          end
operators global
  FOR EACH  Id :  identifier
  SUCH THAT Id := extract(Vars) : {
    Id : poly_variable;
  }
  ...

specification someVariables
  Vars  X.Y.Z.nil
end
```

Running the system




```
> elan poly someVariables
```

Comparison with other systems

- EQI (Chicago, M. O'Donnell, 1983): no longer supported
- OBJ3 (Menlo Park, J. Goguen, J. Meseguer, 1987): no longer supported
- Maude (Menlo Park, J. Meseguer, 1996): interpreter
- CafeOBJ (Ishikawa, K. Futatsugi, 1995): interpreter
- ASF+SDF (Amsterdam, P. Klint, 1989): interpreter and compiler

Comparison (cont.)

- OBJ3, CafeOBJ, Maude family:
  - good modularity
  - order sorted specifications
  - A, AC, ACU rewriting
  - innermost and outermost rewriting
- ASF+SDF:
  - no parametrization
  - good syntax definition formalism
  - list matching (A rewriting)

Questions?

Advanced introduction

Labelled rules

In ELAN, there are two kinds of rules:

- unlabelled rules (as previously)
- labelled rules
  - they are not automatically applied
  - their application can be controlled

Exercise

```
rules for term
  x : term;
global
  [R1] f(x) => a end
  [R2] f(x) => b end
  [R3] f(x) => c end
end
```

To which term does $f(a)$ reduce to?

Exercise

```
rules for term
  x : term;
global
  [R1] f(x) => a end
  [R2] f(x) => b end
  [R3] f(x) => c end
end
```

To which term does $f(a)$ reduce to?

a?

Exercise

```
rules for term
  x : term;
global
  [R1] f(x) => a end
  [R2] f(x) => b end
  [R3] f(x) => c end
end
```

To which term does $f(a)$ reduce to?

a?, b? or c?

Answer

- the system is not confluent
- the result depends on the strategy
  - if we try `R1` or else `R2` or else `R3` (strategy first),
    the result is a (strategy first)
  - if we do not mind (strategy dont care),
    the result is either a, either b, either c
  - is we want all possible results (strategy dont know),
    the result is a set: {a,b,c}

Elementary strategies

Built from a few built-in constructors:

- Rule:
  R
- Sequential composition:
  S_1 ; S_2
- Iteration:
  repeat(S) or iterate(S)
- Choice:
  dk$(S_1, \ldots, S_n)$: all results of all S_i
  dc$(S_1, \ldots, S_n)$: all results of one S_i
  first$(S_1, \ldots, S_n)$: all results of the first S_i

Elementary strategies (cont.)

- Choice (cut):
  dc_one$(S_1, \ldots, S_n)$: one result of one S_i
  first_one$(S_1, \ldots, S_n)$: one result of the first S_i
- Failure: fail
- Identity: id

Example: enumerate integers

```
operators global
  enum(@,@): (int int) list[int] ;
end

rules for list[int]
  i,j: int;
global
  [] enum(i,j) => i.enum(i+1,j) if i<=j    end
  [] enum(i,j) => nil           if i>j     end
end
```

Exercise

What is the result of: `enum(3,6)`?

Exercise

What is the result of: `enum(3,6)`?

`3.4.5.6.nil`

Exercise

What is the result of: `enum(3,6)`?

`3.4.5.6.nil`

Is it possible to generate 3, 4, 5, 6 successively?

Solution: using a strategy

```
rules for int
  i,j: int;
global
  [iter]  enum(i,j) => enum(i+1,j) if i<j    end
  [final] enum(i,j) => i                     end
end
```

How does the strategy look like?

Solution: using a strategy

```
rules for int
  i,j: int;
global
  [iter]  enum(i,j) => enum(i+1,j) if i<j     end
  [final] enum(i,j) => i                      end
end
```

How does the strategy look like?

```
strategies for int
  [] enumStrat => iterate*(iter) ; final      end
end
```

Rules with AC functions

```
operators global
  @ U @       : (set set) set (AC);
  empty       : set;
  [@]         : (int) set;
  extract(@)  : (set) int;
end
rules for int
  i: int; s: set;
global
  [Rule] extract( [i] U s )  => i end
end
```

Application of `[Rule]` to `extract(empty U [1] U [2] U [3])` returns one of `1,2,3`.
Application of the strategy `dk(Rule)` returns all these results successively.

Example: Nqueens revisited

Given an AC operator:

```
@ U @ :  (set set) set (AC);
```

A rule:

```
[extractrule] (i) U S => [i,S] end
```

And a strategy:

```
[] extractPos => dk(extractrule) end
```

We can extract an element from a set

The algorithm can be encoded in two rules

```
[queensrule]  queens(set,sol) => queens(newSet,pos.sol)
              where [pos,newSet] := (extractPos) set
              if    ok(1,pos,sol)
end
[final] queens(Empty,sol) => sol
end
```

and a strategy

```
[] queens => repeat*(dk(queensrule)) ; dc(final) end
```

Run it

```
Query term: queensAC(4) end
1.3.0.2.nil
2.0.3.1.nil
2 solutions

Query term: queensAC(8) end
...
3.1.6.2.5.7.4.0.end
92 solutions
```

Questions?

Comparison with other systems

- Maude:
  - reflexivity (powerful but not easy to use)
- ASF+SDF:
  - matching condition
  - no strategy, but traversal operators
- Stratego:
  - powerful strategy language (with traversal strategies)
  - no efficient normalisation strategy (i.e. computation rules)
  - no equational rewriting
  - no global backtracking

Application: Protocol verification

Needham-Schroeder public-key protocol

- proposed in '78 by Needham and Schroeder
- proved insecure in '95 by G. Lowe
- Needham-Schroeder public-key protocol has been already analyzed using several methodologies
  - model-checkers - FDR [Roscoe94], Murφ[MitchellMS96]
  - theorem proving - NRL [Meadows96]
  - rewriting - Maude [DenkerMT98], ELAN[Cirstea01]

The protocol

The Needham-Schroeder public-key protocol aims to establish a mutual authentication between an initiator and a responder that communicate via an insecure network.

		Initiator	Responder	Net
1.	$A \rightarrow B$: $\{N_A, A\}_{K(B)}$	A^{SLEEP}	B^{SLEEP}	$\emptyset$
		A^{WAIT}	B^{SLEEP}	$\{N_A, A\}_{K(B)}$
2.	$B \rightarrow A$: $\{N_A, N_B\}_{K(A)}$	A^{WAIT}	$B^{\{N_A,A\}_{K(B)}}$	$\emptyset$
		A^{WAIT}	B^{WAIT}	$\{N_A, N_B\}_{K(A)}$
3.	$A \rightarrow B$: $\{N_B\}_{K(B)}$	$A^{\{N_A,N_B\}_{K(A)}}$	B^{WAIT}	$\emptyset$
		A^{COMMIT}	B^{WAIT}	$\{N_B\}_{K(B)}$
4.	N_B *is the session key*	A^{COMMIT}	B^{COMMIT}	$\emptyset$

Methodology

Detect possible attacks of the protocol by the symbolic execution of all the possible states.

Undeterministic strategies are needed to make the description of the search space easier.

Encoding the Needham-Schroeder public-key protocol in ELAN

- user-definable number of initiators and responders
- user-definable size of the network
- the strategies guiding the rewrite rules describe a form of model-checking in which all the possible behaviors are explored

Data structures

- The agent

```
@ + @ + @  : ( AgentId SWC Nonce ) Agent;
@        : ( Agent ) listAgent;
@ || @   : ( listAgent  listAgent ) listAgent (AC);
```

- The messages

```
@-->@:@[@,@,@] : ( AgentId AgentId Key Nonce Nonce Address ) message;
```

- The network

```
@       : ( message ) network;
@ & @   : ( network  network ) network (AC);
```

- The intruders

```
@ # @ # @ : ( AgentId listNonce network ) intruder;
```

- The global state

```
@ <> @ <> @ <> @ : ( listAgent listAgent intruder network ) state;
```

Rewrite rules for the agents

- initiator starts the communication with a responder

```
[initiator-1]
   x+SLEEP+resp  || E <> D <> w#l#ll <> ls                          =>
   x+WAIT+N(x,y) || E <> D <> w#l#ll <> x-->y:K(y)[N(x,y),DN,A(x)] & ls
                 where y+std+init :=(extAgent) elemIA(D || w+SLEEP+DN)
end
```

- responder reads the message and sends the acknowledgement

```
[responder-1]
   E <> y+SLEEP+init  || D <> I <> w-->y:K(y)[N(n1,n3),N(n2,n4),A(z)] & ls =>
   E <> y+WAIT+N(y,z) || D <> I <> y-->z:K(z)[N(n1,n3),N(y,z),A(y)] & ls
end
```

Rewrite rules for the agents

- initiator receives the acknowledgement and checks its validity

```
[initiator-2]
 x+WAIT+N(x,v) || E <> D <> I <> w-->x:K(x)[N(n1,n3),N(n2,n4),A(z)] & ls =>
 S
  choose
    try
      if x==n1 and v==n3
      where S:=() x+COMMIT+N(x,v) || E <> D <> I <> x-->v:K(v)[N(n2,n4),DN,DA] &
    try
      if x!=n1 or v!=n3
      where S:=() ERROR
  end
end
```

Rewrite rules for the intruder

- the intruder intercepts all the messages in the network but the messages generated by itself and stores or decrypts them.

```
[intruder-1]
   E <> D <> w#l#ll <> z-->x:K(w)[N(n1,n3),N(n2,n4),A(v)] & ls  =>
   E <> D <> w#N(n1,n3) | N(n2,n4) | l#ll <> ls
                 if w!=z
end
```

- the nonces obtained previously by the intruder are used in order to generate fake messages that are sent to all the agents.

```
[intruder-4]
   E <> D <> w # resp | l # ll <> ls                              =>
   E <> D <> w # l # ll <> w-->y:K(y)[resp,DN,A(xadd)] & ls
                where y+std+dn :=(extAgent) elemIA(D || E)
                where xadd+std1+dn1 :=(extAgent) elemIA(D || E)
end
```

The invariants

- authenticity of the responder: if an initiator x committed with a responder y, then y has really been involved in the protocol.

```
[attack-1] x+COMMIT+N(x,y) || E <> D <> I <> ls =>
           ATTACK
                if y!=i
                if not(existAgent(y+WAIT+N(y,x),D)) and
                   not(existAgent(y+COMMIT+N(y,x),D))
end
```

- authenticity of the initiator: if a responder `y` committed with an initiator `x` then the initiator have committed as well with `y`.

```
[attack-2] E <> y+COMMIT+N(y,x) || D <> I <> ls =>
           ATTACK
                if x!=i
                if not(existAgent(x+COMMIT+N(x,y),E))
end
```

The strategy

We apply repeatedly all the rewrite rules in any order and in all the possible ways until one of the attack rules can be applied.

```
[]attStrat  =>  repeat*(
                   dk(
                      attack-1, attack-2,
                      intruder-1, intruder-2, intruder-3, intruder-4,
                      initiator-1, initiator-2, responder-1, responder-2)
                );
                attackFound
end
```

where

```
[attackFound]   ATTACK   =>   ATTACK              end
```

The attack

I tries to impersonate A to establish a session with B.
2 simultaneous runs of the protocol.

I.1.	$A \to I$	: $\{N_A, A\}_{K(I)}$
II.1.	$I(A) \to B$	: $\{N_A, A\}_{K(B)}$
II.2.	$B \to I(A)$	: $\{N_A, N_B\}_{K(A)}$
I.2.	$I \to A$	: $\{N_A, N_B\}_{K(A)}$
I.3.	$A \to I$	: $\{N_B\}_{K(I)}$
II.3.	$I(A) \to B$	: $\{N_B\}_{K(B)}$

The corrected protocol

1. $A \to B$: $\{N_A, A\}_{K(B)}$
2. $B \to A$: $\{N_A, N_B, B\}_{K(A)}$
3. $A \to B$: $\{N_B\}_{K(B)}$

Modified rule

```
[initiator-2]
x+WAIT+N(x,v) || E <> D <> I <> w-->x:K(x)[N(n1,n3),N(n2,n4),A(z)] & ls  =>S
 choose
  try
   if v==z               // expected responder
   if x==n1 and v==n3
   where S:=
         () x+COMMIT+N(x,v) || E <> D <> I <> x-->v:K(v)[N(n2,n4),DN,DA] & ls
  try
   if v!=z               // not expected responder
   if x!=n1 or v!=n3
   where S:=() ERROR
  end                                                                        end
```

Comparing to existing approaches

- we explore a finite state space and we can consider it as a form of model-checking,
- in the Maude approach an agent can participate simultaneously in several sessions.
- comparison with Murφ - a general state enumeration tool

Comparison with Murφ

ini	res	net	Murφ	ELAN simple	ELAN(Murφ like)	ELAN optimized
1	1	1	2578 rules 0.56s	7442 rules 0.167s	4308 rules 0.125s	711 rules 0.064s
1	1	2	136273 rules 18.40s	453514 rules 7.007s	333552 rules 4.980s	6700 rules 0.162s
2	1	1	25701 rules 6.81s	575101 rules 8.571s	257087 rules 3.946s	26011 rules 0.483s
2	2	1	557430 rules 303.46s	118214389 rules 1658.296s	22985807 rules 333.096s	753785 rules 12.392s