HAL open science

# Objects, rules and strategies in ELAN

Hubert Dubois, Hélène Kirchner

## To cite this version:

Hubert Dubois, Hélène Kirchner. Objects, rules and strategies in ELAN. 2nd AMAST Workshop on Algebraic Methods in Language Processing - AMILP'2000, 2000, Iowa City, Iowa, USA, 19 p. inria-00099054

**HAL Id: inria-00099054**

**https://inria.hal.science/inria-00099054v1**

Submitted on 26 Sep 2006

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

# Objects, rules and strategies in ELAN

Hubert Dubois and Hélène Kirchner
LORIA-UHP & CNRS
BP 239
54506 Vandœuvre-lès-Nancy Cedex, France
{Hubert.Dubois|Helene.Kirchner}@loria.fr

**Abstract**

This paper gives an introduction to the ELAN rule-based programming language and presents a general approach for prototyping a new language by transformations defined in ELAN. The approach is used to design an extension of ELAN with objects, leading to an expressive programming framework that combines the concepts of objects, rules and strategies.

**Keywords:** Rule-based programming, rewrite rule, strategy, object.

## 1 Introduction

Rules are ubiquitous in Computer Science and appear in various contexts under several terminologies: production rules, grammar rules, transition rules, inference rules, type checking rules, to cite a few. On one hand, rule-based systems are largely used in the artificial intelligence community, on the other hand, in the programming languages area, the concept occurs through logic programming rules, functional programming rules, constraint handling rules, program transformation rules,... However one may distinguish two main kinds of rules: computation rules whose purpose is to compute as fast as possible the unique result, and deduction rules whose application needs to be controlled. This control can be expressed through various means, using concepts such as search plans, action plans, tactics, occurring in expert systems or theorem provers, or lazy evaluation, innermost/outermost reduction, and so on, introduced for evaluation of programming languages. Note that in most programming languages, the evaluation strategy is fixed. Although the evaluation process is then easier to implement when the strategy is fixed, this rigidity is inconvenient when the strategy wanted by the user is different from the built-in one. Programmers have then to design special data structures that can become complex to express the desired strategy. We advocate the prime interest of a strategy language that allows us to separate control from logic and data structures, to express the control easily and in a declarative setting, and to reason about strategies. We study, implement and experiment these ideas in ELAN.

The first goal of this paper is to present the ELAN language and the programming style provided by rules and strategies. It is not a new idea that rule based programming is well-suited to programming language processing. So it is not quite surprising to apply ELAN to this area. A first application domain of our system is thus to easily prototype a new language by transformations defined in ELAN, and using ELAN as target for the translation. This approach is first illustrated on Prolog programs translated into ELAN.

The second goal of this paper is to present an extension of the ELAN language with objects and some features of object-oriented programs. Object oriented programming provides a high-level formalism to represent structured data, whose components can be selected via attributes, and states evolving along the time or thanks to external processes. Objects are instances of classes which share the same structure, the same methods and inheritance properties. This language extension has been prototyped using the same approach based on programming language translation in ELAN. The language is enriched by a notion of objects and object modules (OModules for short) which are translated respectively into structured terms and into ELAN modules. Finally we show how objects, rules and strategies can be combined

to achieve an expressive programming language. The design of a multi-elevator controller using these concepts illustrates this last point.

The paper is organised as follows: Section 2 presents the language ELAN with the concepts of rules and strategies. Section 3 shows how to use ELAN in order to transform a program in another language that we want to prototype into an ELAN program that can be executed. Section 4 proposes an extension of ELAN with objects and applies the previous method of Section 3 to prototype this extension. Section 5 shows how to incorporate objects, rules and strategies in the same environment.

## 2 ELAN

The ELAN system (Borovanský et al., 1996), (Borovanský et al., 1998) provides a very general approach to rule-based programming. ELAN offers an environment for specifying and prototyping deduction systems in a language based on rewrite rules controlled by strategies. It offers a natural and simple logical framework for the combination of computation and deduction paradigms. It supports the design of theorem provers, logic programming languages, constraint solvers and decision procedures and offers a modular framework for studying their combination. ELAN has a clear operational semantics based on rewriting logic (Borovanský et al., 1999). Its implementation involves compiled matching and reduction techniques integrating associative and commutative functions. Non deterministic computations return several results whose enumeration is handled thanks to a few constructors of choice strategies. A strategy language is available to control rewriting. Evaluation of strategies and strategy application is again based on rewriting.

The language is close to the algebraic specification formalism but provides additional specificities that are worth emphasising. Three main principles have guided the design of the ELAN language.

- First, the language allows rules to be non-terminating and non-confluent, but then their application has to be controlled. According to the distinction made in the introduction, there are rules for computations, which are required to be confluent and terminating, in order to give a unique result, and rules for deductions, for which no confluence nor termination is required.

- Rules and strategies are first-class objects in the language A strategy language is provided to express control and derivation tree exploration. A few strategy constructors, similar to those for tactics in proof assistants, are offered and efficiently implemented, but the user may also design his own strategies.

- Application of rule or a strategy on a term may give 0, 1, or several results. This non-determinism related to the production of sets of results is handled by backtracking.

As a consequence of these features, the language allows different programming styles. Functional programs are naturally expressed with confluent and terminating rules, while the backtracking mechanism used to handle several results gives a flavour of logic programming and allows to program non-determistic computations. The main originality is surely the capability of strategy programming for expressing the control of programs in a declarative way.

### 2.1 Modules

Following the algebraic languages tradition, ELAN is modular. A program is a collection of hierarchically constructed modules together with a request, which is a term to be evaluated in this hierarchy. A module may import already defined modules and the type of this importation may be local (not visible outside the module) or global (visible outside). A module defines also a set of sorts, a list of operators with their types, several lists of rules classified by the type of their left and right-hand sides (lhs and rhs for short), and strategies also defined by operators and rules.

In order to progressively introduce all these ingredients, let us first consider a simple module (Fig. 1) which defines an operator *enum* that takes two integer arguments and returns the list of integers greater than the first and less than the second argument.

Predefined modules exist in the ELAN library, such as `bool, int, ident, list[X]`... Here `int` and `list[int]` are imported and provide the sorts `int` and `list[int]`.

```
module enum0
import  global int list[int] ; end

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

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

Figure 1: A simple module in ELAN

## 2.2 Confluent and terminating rules

The module given in Fig. 1 illustrates how conditional rewrite rules are grouped together according to the sort of their left (or right) hand side. Because of the two mutually exclusive conditions in the rules, this rewrite system is confluent and terminating. An application of this set of rules on an initial term, say `enum(3,6)`, produces a unique result, here `3.4.5.6.nil`.

## 2.3 Non-deterministic computations

Now we could ask the following question: instead of having `enum(3,6)` as a list `3.4.5.6.nil` is it possible to generate successively `3, 4, 5, 6`? In other words, can we write non-deterministic programs? The answer is indeed affirmative, but this requires to change the programming style. Let us detail this point on the example. First, to achieve that goal, the type of `enum` is changed to return an integer, and a label is given for each rule: let us call `[final]` the rule `enum(i,j) => i` which returns the first argument of `enum` and `[iter]` the recursive rule `enum(i,j) => enum(i+1,j) if i<j`.

Then, a strategy operator `enumStrat1` is defined by a rewrite rule on a strategy sort `<int->int>` (implicit as soon as the sort `int` is declared). The strategy is expressed as the iteration of the recursive rule `[iter]` that returns intermediate results of the form `enum(i,j)` concatenated with the rule `[final]` which returns `i`. The whole program is shown in Fig. 2.

An application of the strategy `enumStrat1` on an initial term, say `(enumStrat1)enum(3,6)`, produces four results `3, 4, 5, 6`.

```
module enum1
import  global int ; end

operators global
  enum(@,@): (int int) int ; end

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

end

stratop global
  enumStrat1 : <int->int>          ; end

strategies for int
[] enumStrat1 => iterate*(iter) ; final  end
end
```

Figure 2: A module with strategies

## 2.4 Constructors for strategies

On the simple example of Fig. 2, we have already seen how to define an elementary strategy. In a more general way, a first class of strategies can be built from the labelled rules and from a few built-in constructors: sequential composition, iterators, choice points, cut points, failure and identity.

Formally, a strategy is a function which, when applied to an argument, returns a set of possible results. The strategy fails if the set is empty. In order to provide the user with the capability to easily specify the control, a strategy language offers the following strategy constructors:

- A labelled rule is a primal strategy. Applying a rule labelled lab returns in general a set of results. This primal strategy fails if the set of results is empty.

- Two strategies can be concatenated by the symbol "**;**", i.e. the second strategy is applied on all results of the first one. $S_1$ **;** $S_2$ denotes the sequential composition of the two strategies. It fails if either $S_1$ fails or $S_2$ fails. Its results are all results of $S_1$ on which $S_2$ is applied and gives some results.

- **dk**$(S_1, \ldots, S_n)$ chooses all strategies given in the list of arguments and for each of them returns all its results. This set of results may be empty, in which case the strategy fails.

- **first**$(S_1, \ldots, S_n)$ chooses in the list the first strategy $S_i$ that does not fail, and returns all its results. This strategy may return

more than one result, or fails if all substrategies $S_i$ fail.

- **first_one**$(\mathbf{S_1}, \ldots, \mathbf{S_n})$ chooses in the list the first strategy $S_i$ that does not fail, and returns its first result. This strategy returns at most one result or fails if all substrategies fail.

- The strategy **id** is the identity that does nothing but never fails.

- **fail** is the strategy that always fails and never gives any result.

- **repeat**$*(\mathbf{S})$ applies repeatedly the strategy $S$ until it fails and returns the results of the last unfailing application. This strategy can never fail (zero application of $S$ is possible) and may return more than one result.

- The strategy **iterate**$*(\mathbf{S})$ is similar to **repeat**$*(\mathbf{S})$ but returns all intermediate results of repeated applications.

The easiest way to build a strategy is to use the strategy constructors to build strategy terms and to define a new constant operator that denotes this (more or less complex) strategy expression. This gives rise to a first class of strategies called elementary strategies. As illustrated in Fig. 2, elementary strategies are defined by unlabelled rules of the form $[]\ S \Rightarrow strat$, where $S$ is a constant strategy operator and *strat* a term built on predefined strategy constructors and rule labels, but that does not involve $S$. The application of a strategy $S$ on a term $t$ is denoted $(S)\ t$.

Another example given in Fig. 3 for the use of strategy construction is the representation of a tree structure and the strategy to go through this tree.

The strategy `allleaves` applies on the tree `node(node(leaf(1),2,leaf(3)),4,node(leaf(5),6,leaf(7)))` returns the set of results `1, 3, 5, 7`.

## 2.5 Rules with local variables and patterns

Labelled rules and more generally strategies are always applied at the top position of a term. In order to be able to apply them inside expressions, a more general form of rule is allowed in ELAN, with local variables allowing to apply

```
module tree
import global int ; end

sort int tree ; end

operators  global
        @              : (int) tree;
        leaf(@)        : (int) tree;
        node(@,@,@)    : (tree int tree) tree;
end

rules for tree
        l, r: tree;  n: int;
global
[Left]     node(l,n,r)    => l              end
[Right]    node(l,n,r)    => r              end
[L_value]  leaf(n)        => n              end
end

stratop
  global
        allleaves  : <tree -> tree>   ;    end

strategies for tree
  [] allleaves  => iterate*(dk(Left,Right));L_value end
end
end
```

Figure 3: A tree exploration with strategies

strategies on subterms. Let us consider a program for polynoms derivation with respect to one variable `x`. Assume that it defines a strategy `simplify` that puts any polynom in some canonical form, for instance a sum of monomials of decreasing degree w.r.t. `x`. When the derivation of a product in defined, one may want to put each component of the resulting sum in this canonical form. This is possible thanks to the introduction of local variables that register intermediate computations. This is illustrated in Fig. 4.

```
rules for poly
  p1, p2, p3, p4, p5 : poly; x variable;
global
  []  deriv(p1*p2,x)   => p5
        where p3:=(simplify) deriv(p1,x)*p2
        where p4:=(simplify) p1*deriv(p2,x)
        where p5:=(simplify) p3+p4 end
end
strategies for poly
  [] simplify => repeat*(first one(expand)) ;
                 repeat*(first one(factorize))
end
```

Figure 4: Rules with local variables

One can also generalise variables to patterns, i.e. terms with variables and write for instance a pair `<x,y>` to register the two components of the position of a point, as in Fig. 5.

```
[] move(S) => C(x,y)
      where < x , y > := position(S)
      if x=y
```

Figure 5: Rules with patterns

To summarise, the general form of ELAN rules is actually as follows:

$$[\ell]\ \ l \to r \ \textbf{where}\ \ p_1 := (S_1)c_1 \ldots\ \textbf{where}\ \ p_n := (S_n)c_n$$

- $l, r, p_1, \ldots, p_n, c_1, \ldots, c_n \in \mathcal{T}(\Sigma, \mathcal{X})$,
- $\mathcal{V}ar(p_i) \cap (\mathcal{V}ar(l) \cup \mathcal{V}ar(p_1) \cup \cdots \cup \mathcal{V}ar(p_{i-1})) = \emptyset$,
- $\mathcal{V}ar(r) \subseteq \mathcal{V}ar(l) \cup \mathcal{V}ar(p_1) \cup \cdots \cup \mathcal{V}ar(p_n)$ and
- $\mathcal{V}ar(c_i) \subseteq \mathcal{V}ar(l) \cup \mathcal{V}ar(p_1) \cup \cdots \cup \mathcal{V}ar(p_{i-1})$.

In such expressions, **where** $true := c$ is usually written **if** $c$. The pattern $p_i$ often reduced to a variable $x$. $S_i$ may be the identity strategy, which is written $()c_i$.

To apply the rule

$$[\ell]\ \ l \to r \ \textbf{where}\ \ p_1 := (S_1)c_1 \ldots\ \textbf{where}\ \ p_n := (S_n)c_n$$

to a subject $t$, the matching substitution from $l$ to $t$ ($l\sigma = t$) is successively composed with each matching $\mu_i$ from $p_i$ to $(S_i)c_i\sigma\mu_1 \ldots \mu_{i-1}$, for $i = 1, \ldots, n$. To evaluate each $(S)c$, $c$ is first normalised using the unlabelled rules, then one tries to apply a labelled rule according to the strategy $S$. Choice points are set when there are several results and if at some point the set of results in empty, the system backtracks to the previous choice point. When the rule contains a sequence of matching conditions, failing to satisfy the $i$-th condition causes a backtracking to the previous one.

## 2.6 Associative Commutative functions

Associative Commutative functions introduce an intrinsic non-determinism. Since an AC matching problem can have several solutions, one may want to get all solutions of an AC matching problem and build all possible results of rewriting with these different matching substitutions. The program in Fig. 6 shows how to define sets in ELAN, with an Associative Commutative operator `U`, and how to enumerate elements of a set, using and `extract` operation and the of non-deterministic choice strategy `dk`.

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

Figure 6: AC operators

Application of the rule `[Rule]` to `extract(emptySet U (1) U (2) U (3) U (4) U (5))` returns one of `1,2,3,4,5`. Application of the strategy `dk(Rule)` returns all these results successively.

When an ELAN rule has a left-hand side $l$ or a pattern $p$ that contains AC function symbols, AC matching is called and can return several solutions. this provides an additional potentiality of backtracking.

## 2.7 Defined Strategies

The class of strategies introduced above does not allow to define strategies with parameters, nor strategies which are recursive. The class of defined strategies is devoted to these purposes.

A typical example of a strategy operator taking a parameter, i.e. another strategy as argument is the `map` function. It is declared in ELAN as: `map(@):(<X->X>)<list[X]->list[X]>` with a parameter sort `X`. Its definition is just one rule: `map(s)=>dc(nil,s.map(s))` where `s` is a variable of sort `<X->X>`.

This definition is equivalent to the more classical definition in which we use explicitly the application operator `[@]@`, provided by ELAN. The program in Fig. 7 illustrate the definitions of this function either in its implicit form (`map`) and its explicit form (`mac`).

The application of `[map(mul2)]` and `[mac(mul2)]` on the list of integers `1.2.3.nil` returns indeed both `2.4.6.nil`.

Recursive and parameterised strategies may thus be defined and more examples can be found in (Borovanský, 1998).

```
module map[X]

import global
  strat[X] strat[list[X]] X list[X]; end
stratop global
  map(@)  : (<X->X>) <list[X]->list[X]>;
  mac(@)  : (<X->X>) <list[X]->list[X]>;
end

rules for X
 x : X;  n,m : int;
global
  [mul2] x => x*2               end
end

strategies for list[X]
 s  : <X->X>;
 h  : X;
 h1 : X;
 t  : list[X];
 t1 : list[X];
explicit
[.] [mac(s)]nil => nil                 end
[.] [mac(s)]h.t => h1.t1
       where h1:=[s]h
       where t1:=[mac(s)]t             end
implicit
[.] map(s) => dc(nil,s.map(s))   end
end
end
```

Figure 7: Defined strategies

## 3 A GENERAL TRANSFORMATION SCHEME OF PROGRAMS

In this section, we argue that ELAN can be used to prototype other programming languages, in a systematic and flexible way. We show how to use ELAN to produce, from a program given in the syntax of the language we want to prototype, an ELAN program which can be interpreted and compiled.

The first step consists of reading the input syntax and an input program written in this syntax. The ELAN parser checks that the input program is well-formed with respect to the input syntax and produces a well-typed term representing the input program. In order to manipulate and transform this term, a specific structure is designed, giving an internal representation of the input program. The second step of the transformation produces this internal representation. A third step consists of rewriting this presentation of the input program into an internal representation of an ELAN module, according to a pseudo ELAN syntax and to produce, in a fourth step, a corresponding file. The last step just corrects syntax approximations and generates a readable file which can be parsed by the ELAN parser. This transformation is entirely written in ELAN from the first step to the fourth one. The last step uses shell script with `sed` commands to produce a more readable output program and to transform a few reserved keywords that ELAN cannot produce himself. The transformation uses unlabelled rules, no user-defined strategy, and from an initial term which matches the input program, it produces a new program in ELAN syntax. This transformation always gives one and only one result, provided that the input program follows the syntax that specifies the language to prototype. The five steps of the transformation are represented in Fig. 8.

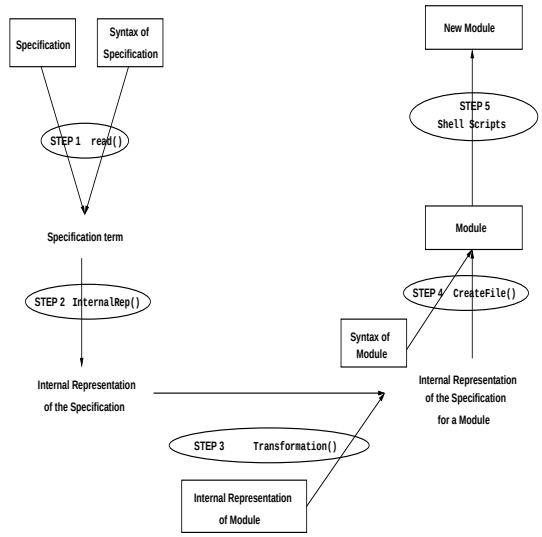



Figure 8: From a specification to ELAN module

In order to illustrate the approach, let us consider a small example of a Prolog program (Fig. 9).

```
Specification Family

Vars       X Y Z
Ops
Predicates father:2 male:1 female:1 grandfather:2
           brother:2 sister:2
Clauses father(John,Ann) :- .
        father(Bill,John) :- .
        father(John,Al) :- .
        male(Al) :- .
        female(Ann) :- .
        grandfather(X,Z) :- father(X,Y),father(Y,Z) .
        brother(Y,Z):-father(X,Y),father(X,Z),male(Y) .
        sister(Y,Z):-father(X,Y),father(X,Z),female(Y).
End of Specification
```

Figure 9: A specification in Prolog

The specification of a Prolog program is given by a list of variables, a list of operators with

their arity, a list of predicates with their arity and a list of clauses. A program written in this syntax is transformed in order to produce an ELAN module. A Prolog goal is then parsed by ELAN in this module and evaluated thanks to an ELAN program that implements the SLD-resolution.

In order to program the transformation in ELAN, an operator `Transformation` is defined, which takes as unique argument the name of the file to be transformed, and which returns `true` if the transformation succeeds. The rule defining this operator is the top level of the transformation process and is expressed as follows:

```
[] Transformation(file) => true
  where Term1 := () GetTermFromSpecif(file)
  where Term2 := () InternalRepOfSpecif(Term1)
  where Term3 := () FromInternalRepToNewModule(Term2)
  where Module := () CreateNewModule(Term3)
end
```

The input file is given as argument to the operator `Transformation` which returns `true` if the transformation succeeds and which produces intermediate terms `Term1`, `Term2`, `Term3` and `Module` corresponding to the respective results of each step of the transformation. Let us now describe more precisely how these four first steps are written in ELAN.

## 3.1 Reading the input program

The first step presented in Fig. 10 corresponds to reading the input program.

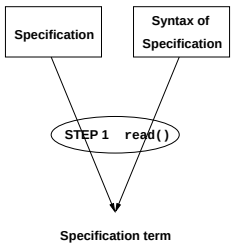



Figure 10: Reading the input program

The operator `GetTermFromSpecif` defined for reading the specification uses the input primitive `read` offered by ELAN, and is defined as follows:

```
[] GetTermFromSpecif(file)     => pg
    where pid := () open(file,"r")
    where pg := () read(pid)
    where pid1 := () close(pid)
end
```

In order to read and parse this term, we have also to define in an ELAN module the syntax of the program given as input. This module gives the grammar of the programming language we want to prototype. For the Prolog example, this syntax is described in an ELAN module whose operators declarations are given in Fig 11.

```
operators       global

  Specification @ @ End of Specification :
          (identifier Spec:SpecifParts) Specif;

  Vars @ Ops @ Predicates @ Clauses @    :
          (V:SpecifVars
           O:SpecifOps
           P:SpecifPreds
           C:SpecifClauses) SpecifParts;

  @         : (identifier) SpecifVars;
  @ @       : (identifier SpecifVars) SpecifVars;

  @ ':' @   : (identifier int) SpecifOps;
  @ ':' @ @ : (identifier int SpecifOps) SpecifOps;

  @ ':' @   : (identifier int) SpecifPreds;
  @ ':' @ @ : (identifier int SpecifPreds)SpecifPreds;

  @ .       : (Clause) SpecifClauses;
  @ . @     : (Clause SpecifClauses) SpecifClauses;

  @         : (Atom) Clause;
  @ ':'- @  : (Atom ListAtoms) Clause;

  @         : (Atom) ListAtoms;
  @ , @     : (Atom ListAtoms) ListAtoms;

  @         : (term) Atom;
  @         : (equation) Atom;

  @         : (identifier) term;
  @(@)      : (identifier Subterms) term;

  @         : (Subterm) Subterms;
  @ , @     : (Subterm Subterms) Subterms;

  @         : (term) Subterm;
  @         : (identifier) Subterm;

  @ = @    : (term term) equation;
end
```

Figure 11: Prolog grammar in ELAN

Using this ELAN file, and given the file presented in Fig. 9, the ELAN parser builds the following term of sort `Specif`.

```
Specification Family Vars X Y Z Predicates
father:2male:1female:1grandfather:2brother:2
sister:2Clauses father(John,Ann).father(Bill,John).
father(John,Al).male(Al).female(Ann).
grandfather(X,Z):-father(X,Y),father(Y,Z).
brother(Y,Z):-father(X,Y),father(X,Z),male(Y).
sister(Y,Z):-father(X,Y),father(X,Z),female(Y).
End of Specification
```

So, if this step succeeds, we get a term in the syntax defined by the user. This ensures that the program is well-formed with respect to the syntax of the new language. A non well-formed

program cannot be read and the transformation process stops at its first step without evaluating the others.

## 3.2 Internal representation of the input term

In order to get a more structured form and to give a better access to the different components of the program, an internal representation is defined. The second step of the transformation is represented in Fig. 12.

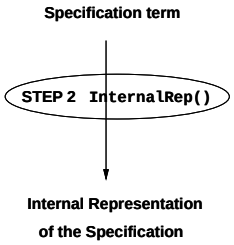



Figure 12: Internal representation of the input term

This internal structure takes advantage from data structures predefined in ELAN like lists, pairs, n-tuples. For instance, the internal representation of the Prolog program is composed of four parts, corresponding to the declaration of variables, operators, predicates and clauses. It is represented as a 4-tuple, whose first component is a list of identifiers (one identifier for each variable); the second one is a list a pairs *[name of the operator,arity]* to encode the list of operators defined with their arities; the third component is also a list of pairs *[name of the operator,arity]* for the operators and the last component is a list of pairs *[clause in the lhs,list of clauses in the rhs]*.

The operator used for this step is called `InternalRepOfSpecif`. From the input term, it produces the following internal representation:

```
[X.Y.Z.nil,
nil,
[father,2].[male,1].[female,1].[grandfather,2].
  [brother,2].[sister,2].nil,
[father(John,Ann),nil].
  [father(Bill,John),nil].
  [father(John,Al),nil].
  [male(Al),nil].
  [female(Ann),nil].
  [grandfather(X,Z),father(X,Y).father(Y,Z).nil].
  [brother(Y,Z),father(X,Y).father(X,Z).male(Y).nil].
  [sister(Y,Z),father(X,Y).father(X,Z).female(Y).nil].
  nil]
```

Note that some parser can directly give this structured representation under the form of annotated terms (ATerms) with primitives giving access to their different components (Van den Brand et al., 2000).

## 3.3 Transformation of the internal representation

In the next step, presented in Fig. 13, the internal representation of the initial program is transformed into an internal representation of an ELAN program which is composed of a module name, the list of imported modules, the list of sorts defined in the module, the list of operators, rules and strategies.

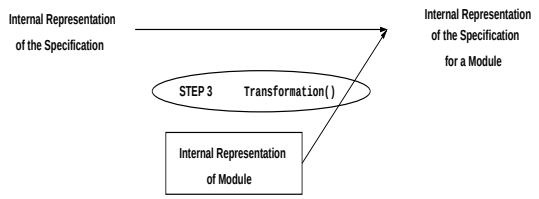



Figure 13: Correspondence between two internal representations

The transformation from an internal transformation into another one is performed by ELAN rules. This is essentially an enrichment by new operators and new rules deduced automatically form the previous internal representation. For instance, in the case of Prolog program, a clause from the input Prolog program is a subterm of the internal representation. In order to mimic SLD-resolution, the corresponding ELAN program must have access to these subterms, to their head symbol, to their subterms. All these operations have to be added to the representation. The term corresponding to the specification given in Fig. 9 is partially given below.

```
[Prolog,
nil,
nil,
 [[X,0],[nil,variable]].
 [[Y,0],[nil,variable]].
...
 [[female,0],[nil,Psymbol]].
 [[female,1],[atom.nil,term]].
 [[grandfather,0],[nil,Psymbol]].
 [[grandfather,2],[atom.atom.nil,term]].nil,
[term,[t_ 2,term].[t_1,term].nil,
 [NOIDENT,2-th subterm(brother(t_ 1,t_ 2)),t_ 2].
 [NOIDENT,1-th subterm(brother(t_ 1,t_ 2)),t_ 1].nil].
...
[Psymbol,[t_ 2,term].[t_1,term].nil,
 [NOIDENT,head(brother(t_ 1,t_ 2)),brother].nil].
...
```

```
[list[pair[atom,list[atom]]],nil,
 [NOIDENT,clpProg,[father(John,Ann),nil].
 [father(Bill,John),nil].
 [father(John,Al),nil].[male(Al),nil].
 [female(Ann),nil].
 [grandfather(X,Z),father(X,Y).father(Y,Z).nil].
 [brother(Y,Z),father(X,Y).father(X,Z).male(Y).nil].
 [sister(Y,Z),father(X,Y).father(X,Z).
                              female(Y).nil].nil].
 nil].
nil]
```

The different components in this term can be identified: the name of the module `Prolog`, empty lists for imported modules and new sorts, a list for operator definitions and a list of rules where each rule is a 4-tuple: the first component is the name of the rule, followed by the list of variable (a list of pairs *[variable name,sort]*) and, finally, a list of rule bodies (a 3-tuple *[name of the rule,lhs,rhs]*).

## 3.4 Generation of a new module

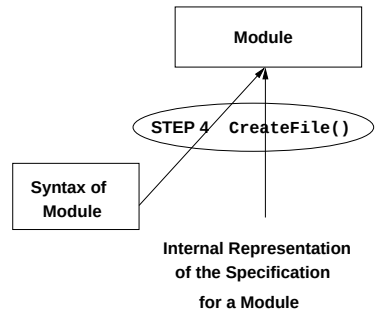
Once the internal transformation of new modules is obtained, it remains to produce the corresponding module in a pseudo ELAN syntax. It is called pseudo ELAN syntax because the ELAN system, due to its current parser, cannot generate some of its own reserved keywords like, for instance, `rules`, `end`, etc.

The target syntax of ELAN modules is needed in order to produce, from the internal representation, a module following this syntax. Again, this syntax is given in an ELAN module, whose top level is described in Fig. 14.

```
module elanModule
import global elanImports elanSortDefinition
              elanOperatorDefinition
              elanStrategyDefinition elanFamilyOfRules
              elanFamilyOfStrategies;
end
sort Module FormalModuleName IdentifierList;
end
operators global
  module @ @ @ @ @ @ @ End :
        (FormalModuleName ImportsOpt
         SortDefinitionOpt OperatorDefinitionOpt
         StrategyDefinitionOpt ListOfFamilyOfRules
         ListOfFamilyOfStrategies) Module;

  @           : (identifier) FormalModuleName;
  @ '[' @ ']' : (identifier IdentifierList)
                              FormalModuleName;

  @       : (identifier) IdentifierList;
  @ ',' @ : (identifier IdentifierList)
                          IdentifierList;
end
end
```

Figure 14: ELAN module syntax

The generation of the new module processed at this step is schematised as follows:



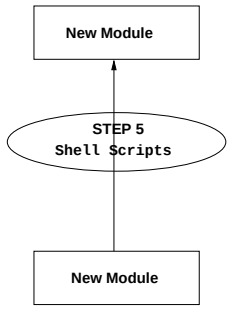

The corresponding module for the Prolog example is:

```
module Prolog

operators global
   X:variable;
   Y:variable;
...
   female:Psymbol;
   female(@):(atom)term;
   grandfather:Psymbol;
   grandfather(@,@):(atom atom)term;
End
...
Rules for term
   t_ 2:term;
   t_1:term;
   NOVAR
global
[]2-th subterm(brother(t_ 1,t_ 2))=>t_ 2 End
[]1-th subterm(brother(t_ 1,t_ 2))=>t_ 1 End
NORULE End

Rules for Psymbol
   t_ 2:term;
   t_1:term;
   NOVAR
global
[]head(brother(t_ 1,t_ 2))=>brother End
NORULE End
...
Rules for list[pair[atom,list[atom]]]
   NOVAR
global
[]clpProg=>[father(John,Ann),nil].
            [father(Bill,John),nil].
            [father(John,Al),nil].
            [male(Al),nil].
            [female(Ann),nil].
            [grandfather(X,Z),father(X,Y).
                              father(Y,Z).nil].
            [brother(Y,Z),father(X,Y).
                          father(X,Z).
                          male(Y).nil].
            [sister(Y,Z),father(X,Y).
                         father(X,Z).
                         female(Y).nil] End
NORULE End
End
```

This step is the last step of the transformation which can be written in ELAN. At this point the obtained module is in a syntax very close to the ELAN one.

### 3.5 A NEW ELAN MODULE

The last step is not written in ELAN but uses shell scripts with mainly the `sed` command.




This operation just corrects syntax approximations and generates a readable file which can be parsed by the ELAN parser. The file is quite similar to the previous one, and, thus, we do not present here the module produced for the Prolog example.

## 4 AN OBJECT LAYER TO ELAN

In this section, we use a similar approach to design an extension of the ELAN language with object-oriented features. In object programming, two kinds of languages can be distinguished: class-based languages like Simula (Birtwistle et al., 1979), C++, Eiffel (Meyer, 1992), Java (Arnold and Gosling, 1996), and object-based languages like Smalltalk (Goldberg and Robson, 1983), where everything is viewed as an object (even the class). The concepts integrated in ELAN are selected features of an object-oriented class-based language: they offer the capability to add object declarations; to define classes of objects with attributes; to add some inheritance properties; to define implicit operators like the access to a field of an object, the creation of an object; to define methods associated to a class of objects.

In order to add these functionalities to ELAN, an object layer is added to the language with proper modules, called OModules, for object modules. The expanded system provides the execution of programs including OModules, mixed with standard ELAN modules. Instead of redesigning an evaluation mechanism for ELAN with objects, OModules are mapped to standard ELAN modules. The transformation method presented in Section 3 is used to perform this mapping.

This section is divided into three parts: the first one describes the structured object layer added to ELAN and the syntax of object modules. The second part defines how each concept can be expressed in rewriting logic, and, more specifically in ELAN. Then, the third part presents an automatic translation of OModules into ELAN modules, which provides the semantics of these OModules in rewriting logic.

### 4.1 A CLASS-BASED LANGUAGE

The definition of a class of structured objects follows the definition of classes in classical object-oriented languages. A class describes a set of objects sharing the same structure and having the same behaviour with respect to methods associated to each class.

Together with a class of objects, a list of attributes is defined, which characterise structured objects of this class. Each attribute has a sort which, when creating an object, is the sort of the value associated to this field. Attributes for a class can be specified as global or local (public or private).

According to the class definition, structured objects are represented as expressions:

$$|Name : Class :: a_1 = v_1, .., a_{N_{Class}} = v_{N_{Class}}|$$

where $Class$ identifies the class (type) of the structured object named by the identifier $Name$, and $(a_1 = v_1, \ldots, a_{N_{Class}} = v_{N_{Class}})$ is the list of pairs (attribute, value) that characterises this object. The order of attributes in the list is irrelevant. All objects in a same class have the same number of attributes $N_{Class}$.

The declaration of attributes automatically creates operator declarations associated to these attributes. For a given attribute $a_i$, these operations are:

- to return the value associated to $a_i$ by the operator $_.a_i$ where $_$ denotes a placeholder for an object. For instance, $Name.a_i$ returns the value $v_i$ associated to the attribute $a_i$ of the object named $Name$.
- to modify the value associated to the attribute $a_i$ by the operator $_[a_i \leftarrow _]$. For instance, associating the value $v_i$ to the attribute $a_i$ of an object $Name$ is denoted by $Name[a_i \leftarrow v_i]$.

For example, a class `Point` with two attributes `X` and `Y` of sort `int` for integer and a

new attribute `Buffer` of sort `int` which is local to this class is defined as in Fig. 15.

```
class Point
imports int
End
attr    X : int
        Y : int
End
```

Figure 15: Definition of attributes

A method is a function which is defined for a given class and which can be applied to any object of the class. A method may be global or local for a given class.

The general definition of a method is given by the sort of its result, a list of variables which can be used, and its body which is a list of instructions consisting in attribute access and modification. In the body of a method, the object to which this method is implicitly applied can be invoked and is denoted by the keyword `self`.

The following example illustrates these notions. Considering the class `Point`, we want to define methods to move a point on a surface: `Up`, `Down`, `Right`, `Left` and `Move(DX,DY)` which add/subtract one unit to `X` or `Y` for the four first methods or moves the point on the surface according to the `DX` and `DY` arguments for the last one. The class `Point` in Fig. 15 with its two global attributes is enriched by these methods in Fig. 16.

```
class Point
imports int
End
attr    X : int
        Y : int
End
methods for Point
  DX,DY : int;
global
Up    <self[Y<-self.Y+1]>
Down  <self[Y<-self.Y-1]>
Right <self[X<-self.X+1]>
Left  <self[X<-self.X-1]>
Move(DX,DY) <self[X<-self.X+DX,Y<-self.Y+DY]>
End
End
```

Figure 16: Definition of methods

In the current definition of class, other classes may be imported. The global attributes defined in the imported classes are added to the current class; this is similar for methods.

The description of the object extension of ELAN is summarised in the definition of the grammar for object modules:

*<OModule>* ::= *<class definitions>*+

*<class definition>* ::= **class** *<class name>*
[*<inherits>*]
[*<imports>*]
[ **attr** [*<global attributes>*]
[*<private attributes>*] **End**]
[*<method defs>*]
**End**

*<inherits>* ::= **from** *<class name>*

*<imports>* ::= **imports** *<list imports* **End**>

*<list imports>* ::= *<module name>*
| *<module name>* ,
*<list imports>*

*<global attributes>* ::= *<attribute declarations>*

*<private attributes>* ::= **private** *<attribute declarations>*

*<attribute declarations>* ::= *<attribute declaration>*
| *<attribute declaration>*
*<attribute declarations>*

*<attribute declaration>* ::=
*<list attribute names>* : *<sort name>*

*<list attribute names>* ::= *<attribute name>*
| *<attribute name>* ,
*<list attribute names>*

*<method definitions>* ::= **methods for** *<sort name>*
*<variable declarations>*
[**private**<*method body*>+**End**]
[**global**<*method body*>+**End**]

*<variable declarations>* ::= *<variable declaration>*
| *<variable declaration>*
*<variable declarations>*

*<variable declaration>* ::=
*<list variable names>* : *<sort name>*

*<list variable names>* ::= *<variable name>*
| *<variable name>* ,
*<list variable names>*

*<method body>* ::= < *<list instructions>* >

*<list instructions>* ::= *<instruction>*
| *<instruction>* ;
*<list instructions>*

*<instruction>* ::= *<attribute modification>*
| *<attribute access>*
| *<affectation>*

*<attribute modification>* ::= *<object name>*
[ *<list modifications>* ]

*<list modifications>* ::= *<modification>*
| *<modification>* ,
*<list modifications>*

*<modification>* ::= *<attribute name>* <- *<NewValue>*

*<attribute access>* ::=*<object name>*.*<attribute name>*

*<object name>* ::=*<identifier>* | **self**

This grammar can be written in an ELAN module by using operator definitions as shown for the Prolog example before.

## 4.2 ELAN to provide an evaluation of OModules

In order to use objects in standard ELAN, the objects modules are translated into ELAN modules. The sort `Object` and the structure of objects are defined in a module `DEFOBJECT`. An object of any class is represented as a term of sort `Object` and is defined as a 3-tuple. The first component, of sort `ObjectIdent`, represents the name of the object, the second component, of sort `ClassName`, is the name of its class, and the third component, of sort `Attribute`, is the list of attributes with the associated values. This list of attributes uses an invisible AC-operator which allows listing the pairs (attribute,value) in any order. The definitions of a value, of an attribute, of a class, of a modification operator, etc..., are given in ELAN parameterised modules, where the parameter is instantiated with relevant instances corresponding to each class of an object-oriented program.

For example, in the design of a planning process for printers management, complex print tasks and simple undecomposable tasks called actions are defined. Tasks have two attributes, here the numbers of items to print and the status to indicate whether to print them together or separately. The class `Task` is defined in Fig. 17.

```
class Task
from Action
imports  TStatus End
attr     NumI   : int
         Status : TStatus
End
End
```

Figure 17: Declaration of class `Task`

The corresponding ELAN module is presented in Fig. 18. This module essentially imports modules parameterised with the corresponding attribute identifier or sorts. The ELAN module `Task` also imports the ELAN module `Action` according to the inheritance definition. It defines one rule that gives the list of attributes for the class `Task`.

## 4.3 Transformation of object modules in ELAN modules

A program in the new language syntax is given in a `file.class` module. An example of such a specification is given in Fig. 17 where the file

```
module Task

import  global
   DEFClass[Task]
   DEFModification[NumI,int]
    DEFModification[Status,TStatus]
   DEFAccess[NumI,int]
    DEFAccess[Status,TStatus]
   TStatus int Action list[AttIdent];
end

rules for list[AttIdent]
global
[] List-Attributes(Task)=>Name.B.E.NumI.Status.nil end
end
end
```

Figure 18: `Task` ELAN module

contains only one class definition. A file containing several class definitions will generate one new module per class. On this example, the transformation process is going to automatically produce only one module presented in Fig. 18. For performing this translation, the syntax of an object module is expressed in ELAN. A part of this ELAN file is shown in Fig. 19.

```
operators       global

// General Syntax of a class

 class @ @ @ @ @ End  : (identifier Inherits Imports
                         Attributes Methods) Class;

// Inheritance definition

 from @          : (identifier) Inherits;

// Importation definition

 imports @ End   : (SortNameList) Imports;

// Attributes declaration

 attr @ @ End    : (GlobalAtts LocalAtts) Attributes;
 attr @ End      : (GlobalAtts) Attributes;

...
end
```

Figure 19: Syntax of OModule in ELAN

The `GetTermFromSpecif()` operator, already defined in Section 3.1, is used to read a module in this syntax. From the OModule in Fig 17, it produces the term:

```
class Task from Action imports TStatus End
 attr NumI:int Status:TStatus End End
```

The second step of the transformation generates, from this term, the following internal representation:

```
[Task,
 Action.nil,
 TStatus.nil,
```

```
[true,Status,TStatus].[true,NumI,int].nil,
nil]
```

of sort:

```
tuple[identifier,
      list[identifier],
      list[Name],
      list[IntRepAttribute],
      list[IntRepMethod]]
```

This term is then translated into another term of sort `list[Module]`, where every element of the list is the representation of a module in a syntax very close to the ELAN one. For our example, the list has only one element, and the result of this transformation is:

```
module Task
import global
  TStatus Action int list[AttIdent]
  DEFClass[Task]
  DEFModification[NumI,int]
  DEFModification[Status,TStatus]
  DEFAccess[NumI,int] DEFAccess[Status,TStatus];
End
operators global Status:AttIdent;NumI:AttIdent;
                 @:(TStatus)Value;End
Rules for list[AttIdent]
global
[]List-Attributes(Task)=>NumI.Status.
                         B.E.Name.nil
End End End.
nil
```

To better understand how this translation is performed, we can show some of the rules which are used to transform the internal representation of the object module into the internal representation of an ELAN module. Let us focus our attention on rules that produce, from the internal representation of attributes declaration, the list of corresponding imported parameterised modules.

First, there is a rule that builds a term representing a module with its various components: its name (MN), a set of importations (MI), a set of operators declarations (MO), and rules definitions (MR). The definition of importation is done by the `GetELANGImports` and `GetELANLImports` operators, respectively dedicated to global and local importations of modules.

```
[] NewClassModules(T) => M.nil
...
    where MI := () import
                    global GetELANGImports(T) ;
                    local  GetELANGLImports(T) ;
                   End
...
    where M := () module MN MI MO MR End
end
```

Then, there are rules for the `GetELANGLImports` operator that deduce the list of global imported modules. Five rules define this operator:

```
rules for ELANListGImports

  I1,I2 : identifier;
  LR    : list[IntRepMethod];
  LI    : list[identifier];
  N     : Name;
  SN    : SortName;
  L     : list[SortName];
  LA    : list[IntRepAttribute];

global

[]GetELANGImports([I1,nil,nil,nil,LR])
  => DEFClass[$I1] list[AttIdent]
end
[]GetELANGImports([I1,I2.LI,nil,nil,LR])
  => $I2 GetELANGImports([I1,LI,nil,nil,LR])
end
[]GetELANGImports([I1,LI,N L,nil,LR])
  => N GetELANGImports([I1,LI,L,nil,LR])
end
[]GetELANGImports([I1,LI,L,[false,I2,SN].LA,LR])
  => GetELANGImports([I1,LI,L,LA,LR])
end
[]GetELANGImports([I1,LI,L,[true,I2,SN].LA,LR])
  =>  DEFModification[$I2,SN] DEFAccess[$I2,SN]
      GetELANGImports([I1,LI,L,LA,LR])
end
end
```

The first rule creates the parameterisation of the module `DEFClass` with the name of the class and also calls a module `list[AttIdent]` which is always needed in the construction. The second rule imports the classes which are inherited and the third one imports the modules which are used in the class definition. The third rule skips local attributes (which are marked by *false* as first element of the 3-tuple) in the list of attributes. The fourth rule handles global attributes (marked by *true* as first element of the 3-tuple) and calls two modules `DEFModification` and `DEFAccess` with their respective parameterisation before calling recursively `GetELANGImports`.

For the last step of the translation, the transformation is realised with the `CreateModule()` operator which takes the internal representation of classes as argument and produces the corresponding file for the set of modules.

Once this file is built, shell scripts are used to transform it and to produce a different ELAN module for each class.

```
module Task

import  global
   DEFClass[Task]
   DEFModification[NumI,int]
    DEFModification[Status,TStatus]
   DEFAccess[NumI,int]
    DEFAccess[Status,TStatus]
   TStatus int Action list[AttIdent];
end

rules for list[AttIdent]
global
[]List-Attributes(Task)=>Name.B.E.NumI.Status.nil end
end
end
```

# 5 Rules and Strategies on Objects

We are now ready to define the data base of objects and their dynamic evolution using rules and strategies. The data base of objects is a multiset of objects representing the current state of the system.

In order to program creation, deletion or modification of objects, rules are quite convenient. They allow in particular to express concurrent modifications on subsets of objects and provide a natural synchronisation mechanism.

Changes in the data base of objects are described through conditional rewrite rules of the form:

$$[lab]\ O_1 \ldots O_k \ \Rightarrow\ O'_1 \ldots O'_m\ \ [\mathbf{if}\ \ t \mid \mathbf{where}\ \ l]^*$$

where $O_1, \ldots, O_k, O'_1, \ldots, O'_m$ are structured objects, $t$ is a boolean term called condition, $l$ a local assignment useful to define auxiliary variables. This rule can have the label $[lab]$, or no label which is denoted $[]$.

Rules are applied to the data base by a rewrite engine that looks for candidates in the set of rules. A rule is candidate if its left-hand side matches a subset of structured objects in the data base of objects. Application of this rule succeeds only if the tests **if** $t$ return *true* and if the local assignments **where** $l$ do not fail.

The data base of objects is then updated, either by modifying instantiated objects occurring in the left-hand side according to their instances in the right-hand side: these objects are called modified objects; or by adding instances of new objects occurring in the right-hand side but not in the left-hand side: these are new objects; or by deleting objects occurring in the left-hand side but not in the right-hand side: we call them deleted objects; finally some objects may appear both in left-hand side and right-hand side without being modified by the rule: these are context objects, which are just checked for being present in the data base. Persistent objects are those objects of the data base that do not occur in the rules.

## 5.1 Translation of objects and rules in ELAN

ELAN provides a tuple constructor $[_, \ldots, _]$ with a flexible arity, allowing the construction of pairs $[_, _]$, 3-tuples $[_, _, _]$, etc. An object is represented as a term $[Name, Class, Attr]$ of sort *Object*, with root operator $[_, _, _]$. The third argument, i.e. the list of attributes, is implemented as a multiset of pairs $[a_i, v_i]$ where $a_i$ is of sort *Attribute* and $v_i$ of sort $Value$. *natt* is a constant denoting the empty list of attributes.

Multisets are built thanks to a binary multiset union operator which is associative and commutative (AC for short). *nobj* is the identity for multiset union. The data base of objects is thus represented as a multiset of objects, that is a term with the root operator $\{_, \ldots, _\}$, which has also a flexible arity.

Thanks to the properties of associativity and commutativity of union on multisets, the order of (attribute, value) pairs in an object is irrelevant, as well as the order of objects in the multiset encoding the data base of objects.

To apply the rules on the data base of objects, they have to be slightly transformed to capture unspecified arguments of AC operators. This is performed by adding new variables in the rewrite rules. This is automatically done in the ELAN interpreter and compiler when the top operator of the left-hand side of a rule is an AC operator. A new variable (denoted $Z$) is added as an additional argument of the top operator $\{_, \ldots, _\}$. Concerning the list of attributes, this is performed by the translation. To capture unspecified (attribute, value) pairs for each object, new variables $At_i$ are introduced for each list of attributes in both sides of rewrite rules.

The rule

$$[lab]\ O_1 \ldots O_k \ \Rightarrow\ O'_1 \ldots O'_m\ \ [\mathbf{if}\ \ t \mid \mathbf{where}\ \ l]^*$$

is automatically transformed into the schema of ELAN rule given in Fig. 20.

In this translation, all objects in the left-hand side of the original rule are of the form $[X_i, Class_i, [a_1, v_1] \ldots [a_{n_i}, v_{n_i}]\ \ At_i]$ and

$Vars$
$X_1, ..., X_k$ : $ObjectIdent$;
$X'_c, X'_m, X'_n, O_1, ..., O_k$ : $Object$;
$At_1, ..., At_k$ : $AttributeList$;
1 $[Lab]\ \{[X_i, Class_i, [a_1, v_1]..[a_{n_i}, v_{n_i}]\ At_i]\}_{i=[1,...,k]}$
$\Rightarrow\ X'_c\ X'_m\ X'_n$
2 **where** $O_j := ()[X_j, Class_j, [a_j, v_j]..[a_{n_j}, v_{n_j}]\ At_j]$
% *for the k objects of the original rule*
3 $[$**if** $t$ | **where** $l]^*$
4 **where** $X'_c := ()\ O_{c\in[1,...,k]}$
% *for each context object*
5 **where** $X'_m := ()\ O_{j\in[1,...,k]}[\{a_l \leftarrow v_l\}_{l\in[1,...,N_j]}]$
% *for each modified object*
6 **where** $X'_n := ()\ [O(n), Class_n,$
$\{[a_{l_1}, v_{l_1}]\}_{l_1\in[1,...,N_n]}$
$\{[a_{l_2}, \perp], \}_{l_2\in[1,...,N_n], l_2\neq l_1}\ natt]$
% *for each created object*

Figure 20: Translation of an object rule.

renamed using local variables $O_i$ (line 2). The lines corresponding to the **if** and **where** statements are reproduced. The objects in the right-hand side are divided into three sets:

- context objects that are in the left-hand side and not changed. They are denoted using variables $X'_c$ (line 4).
- modified objects from the left-hand side that are changed by the rule; they are denoted by variables $X'_m$ (line 5). Each attribute is modified according to the value given in the initial rule.
- new objects that are created by the rule and denoted by new variables $X'_n$ (line 6). Each attribute is initialised either by a value given in the initial rule or by a default value noted $\perp$.

## 5.2 A MULTI-ELEVATOR CONTROLLER

In order to illustrate now the extended language, let us consider a program whose purpose is to automate and control an elevator system for buildings with multiple elevators. To formalise this multi-elevator controller, we define two classes: a class `MLift` for elevators and a class `Call` for the controller.

### The class `Call`

This class describes the central memory for the multi-elevator controller. When people enter the elevator, they select floors where they want to go out. This is formalised by an attribute `LCall` composed of a list of pairs: the first element is the requested floor, and the second one is the list of floors that is selected once people enter the elevator. These are floors that have to be served to unload people.

To distinguish calls that are processed from those that are waiting, we have a second attribute `AssignedCall` composed of calls that have been assigned to an elevator and which are to be processed. This attribute is composed by the same pairs than the attribute `LCall`.

### The class `MLift`

This class describes elevators. Each elevator is an object of this class.

Each elevator is characterised by its current floor (this is the attribute `CF`); its state: is it going up?, down?, or is it waiting for a call? (this is the attribute `State`); its list of floors where it has to stop (the attribute `LStop`).

The sort describing the state of an elevator is called `LiftState`. This sort is defined with two operators: a constant `Wait` of sort `LiftState` and an operator `Move(_)` which takes a term of sort `Direction` as argument (`Up` and `Down` are of sort `Direction`) and returns an element of sort `LiftState`.

We have also three other attributes:

- `Zone` indicates the zone where this elevator is. This attribute is useful if we want to guarantee that when dividing the floors into a number of zones equal to the number of elevators, each zone does not have more than two elevators working at any moment.
- `F` (standing for Flag) whose value is either 0 or 1 indicates that an elevator is working (`F` to 1) or waiting (`F` to 0).
- `I`, standing for Interruption, is an integer which can take value in $\{0, 1\}$ and if `I`$= 0$, then the elevator has no interruption and it is available, if `I`$= 1$, then, the elevator is out of service.

### The rules

Rules defining the main actions on elevators are now described.

The two main rules are the rule `Up` and the rule `Down`. An elevator going upward or downward can continue if the current floor is not a floor occurring in its list of stops or in the list of calls. If the elevator can continue, the current

floor and the zone are updated. A condition to apply these rules is that the value of the flag is 0; this value is updated to 1 after application.

```
[Up] |O1:MLift::State=Move(Up) , F=0 , I=0|
     |O2:LCall|
        =>
      O1[CF<-O1.CF+1,Zone<-NewZone(O1.CF+1),F<-1]
      O2
         if not(in(O1.CF,O1.Stop))
         if not(in(O1.CF,O2.LCall))

[Down] |O1:MLift::State=Move(Down) , F=0 , I=0|
       |O2:LCall|
         =>
        O1[CF<-O1.CF-1,Zone<-NewZone(O1.CF-1),F<-1]
        O2
         if not(in(O1.CF,O1.Stop))
         if not(in(O1.CF,O2.LCall))
```

Each elevator can change its moving direction in two cases: either it has reached the top level (or the bottom level), or its current floor is greater (resp. lower) than the maximum (resp. the minimum) level where it has to stop. This is represented by these two rules `ChangeToDown` and `ChangeToUp`:

```
[ChangeToDown]
    |O1:MLift::State=Move(Up) , F=0 , I=0|
    |O2:Call|
      =>
    O1[State<-Move(Down),F<-1]
    O2
      if O1.CF > Max(O1.LStop) or O1.CF == MaxLevel

[ChangeToUp]
    |O1:MLift::State=Move(Down) , F=0 , I=0|
    |O2:Call|
      =>
    O1[State<-Move(Up),F<-1]
    O2
      if O1.CF < Min(O1.LStop) or O1.CF == MinLevel
```

Each elevator has to stop for different reasons. An elevator stops when its current floor is in its list of requested stops (rule `OpenDoorsStop`) or when it is in the list of calls (rule `OpenDoorsCall`). These rules can be applied in the two moving directions; this corresponds to the variable `S` for the attribute `State`.

If the rule `OpenDoorsStop` is applied, the current floor is removed from the list of stops. If the rule `OpenDoorsCall` is applied, the current floor is also removed from the list of calls and then, the new stops requested by people entering the elevator are added to the list of stops.

```
[OpenDoorsStop]
    |O1:MLift::State=Move(S) , F=0 , I=0|
       =>
    O1[LStop<-removeStop(O1.LStop,O1.CF),F<-1]
       if in(O1.CF,O1.LStop)

[OpenDoorsCall]
    |O1:MLift::State=Move(S) , F=0 , I=0|
    |O2:Call|
       =>
    O1[LStop<-addLStop(O1.LStop,O2.LCall,O1.CF),F<-1]
    O2[LCall<-removeCall(O2.LCall,O1.CF)]
       if in(O1.CF,O2.LCall)
```

If the current floor of an elevator is in the list of calls and in the list of stops, instead of applying consecutively the two previous rules, we just apply one rule labelled `OpenDoorsStopAndCall`.

```
[OpenDoorsStopAndCall]
     |O1:MLift::State=Move(S) , F=0 , I=0|
     |O2:Call|
       =>
     O1[LStop<-addLStop(removeStop(O1.LStop,O1.CF),
                        O2.LCall,O1.CF),F<-1]
     O2[LCall<-removeCall(O2.LCall,O1.CF)]
       if in(O1.CF,O1.LStop)
       if in(O1.CF,O2.LCall)
```

A feature of this multi-elevator controller is that priority is given to a call, and once it has been served, other requested stops are served.

A call is assigned to an elevator whose `State` value is `Wait`. This is done by the rule `AssignACall`. When an elevator can be selected (i.e. there is at least a floor calling an elevator), we compute which floor is selected (this is the nearest one and we call it `NextFloor`) by the function `ChooseNextFloor`. Then, we update the two objects by removing `NextFloor` from the list of calls, by adding it to the list of assigned calls in the central memory and to the list of stops and by choosing the good direction to reach it for the selected elevator.

```
[AssignACall]
    |O1:Call| |O2:MLift::State=Wait , F=0 , I=0|
       =>
    O1[AssignedCall<-AddAssignedCall(O1.AssignedCall,
                        O1.LCall,NextFloor),
       LCall<-removeCall(O1.LCall,NextFloor)]
    O2L[Stop<-NextFloor.nil,F<-1,
        State<-Move(ChooseDir(O2.CF,NextFloor)]
     if not(EqualNil(O1.LCall))
     where NextFloor:=()ChooseNextFloor(O2.CF,O1.LCall)
```

When the elevator reaches a floor, we test if this floor is assigned to it, we apply the rule `OpenDoorsAssignedCall`, that updates the list of stops and the list of assigned calls.

```
[OpenDoorsAssignedCall]
    |O1:MLift::State=Move(S) , F=0 , I=0|
    |O2:Call|
       =>
    O1[LStop<-addLStop(removeStop(O1.LStop,O1.CF),
                       O2.AssignedCall,O1.CF),F<-1]
    O2[AssignedCall<-removeCall(O2.AssignedCall,O1.CF)]
       if in(O1.CF,O2.AssignedCall)
```

A condition to assign a call to an elevator is that at least one elevator must have the attribute `State` to `Wait`. This is possible only

when its list of stops is empty as shown in the rule `Wait`:

```
[Wait] |O1:MLift::State=Move(S) , F=0 , I=0|
         =>
       O1[State<-Wait]
         if EqualNil(O1.LStop)
```

## 5.3 Strategies

In general, the application of a rule on the data base of objects may return several results, for instance when several objects or multisets of objects match its left-hand side. This introduces some non-determinism and the need to control rule application. This is why the concept of strategy is introduced. Strategies are used to control the application of rules on objects: strategies provide the capability to define a sequential application of rules; to make choices between several rules that can be applied; to iterate rules; etc.

For the previous example, we define a few strategies to guide the application of the rules on the data base of objects.

The first one is `ONELIFT` which tries to assign a call to a waiting lift; then, it tries to open the doors of the elevator at current floor if, 1- the floor corresponds to an assigned call, 2- it corresponds to a stop and a call, 3- it corresponds only to a call or 4- only to a stop. If the current floor is not a floor where a stop is required, it looks if the direction of the elevator has to be changed and, otherwise, it continues to go upward or downward.

```
[] ONELIFT   => first(  AssignACall ,
                        OpenDoorsAssignedall ,
                        OpenDoorsStopAndCall ,
                        OpenDoorsCall ,
                        OpenDoorsStop ,
                        ChangeSenseToDown ,
                        ChangeSenseToUp ,
                        Up ,
                        Down)
end
```

This strategy is applied as long as there is an elevator whose flag is not set at 1. To work on a set of elevators, we define the strategy `ONCEALLLIFTS`:

```
[] ONCEALLLIFTS  => repeat*(Wait) ;
                    repeat*(ONELIFT) ;
                    repeat*(RemoveFlag)
end
```

where the rule `RemoveFlag` removes all flags at 1 and put them at 0. To go from an initial situation to a situation where all floors are served and where nobody is waiting in an elevator to go out, we define a main strategy `MAIN` that repeats the rule `Main` until the data base of elevators does not change.

```
[] MAIN => first one (repeat*(Main))
end
```

where the labelled rule `Main` is defined as:

```
[Main] ST => ST1
         where ST1 := (ONCEALLLIFTS) ST
         if ST1 != ST
```

Let us consider an initial situation described as:

```
|O(1):MLift::[CF=2] [State=Wait] [LStop=nil]
            [Zone=0] [F=0] [I=0]|
|O(2):MLift::[CF=11] [State=Wait] [LStop=nil]
            [Zone=1] [F=0] [I=0]|
|O(3):MLift::[CF=14] [State=Wait] [LStop=nil]
            [Zone=1] [F=0] [I=0]|
|O(4):Call::[AssignedCall=nil]
           [LCall=[3,2.6.nil].[9,1.7.nil].
                  [17,8.23.nil].
                  [24,6.8.15.19.nil].nil]|
```

Let us assume that the ground floor is floor 0 and the top level is the level 25. In this initial situation, we have three lifts (`O(1)`, `O(2)` and `O(3)`. The first one is waiting at floor 2, the second at floor 11 and the last one at level 14. Four levels are calling an elevator: the 3rd, 9th, 17th and 24th ones. When an elevator will serve the 3rd floor, then, it will have to stop at floors 2 and 6. For the 9th, 17th and 24th floors, the elevator serving it will also have a list of stops to manage.

If we apply the `MAIN` strategy to this initial term, we have the following execution:

```
'Main:state' :
 |O(1):MLift::[F=0],[LStop=3.nil],[State=Move(Up)],
             [CF=2],[Zone=0]|
 |O(2):MLift::[F=0],[Zone=1],[CF=11],
             [State=Move(Down)],[LStop=9.nil]|
 |O(3):MLift::[F=0],[Zone=1],[CF=14],[State=Move(Up)],
             [LStop=17.nil]|
 |O(4):Call::[AssignedCall=[3,2.6.nil].[9,1.7.nil].
                          [17,8.23.nil].nil],
            [LCall=[24,6.8.15.19.nil].nil]|

'Main:state' :
 |O(1):MLift::[F=0],[LStop=3.nil],[State=Move(Up)],
             [CF=3],[Zone=0]|
 |O(2):MLift::[F=0],[Zone=1],[CF=10],
             [State=Move(Down)],[LStop=9.nil]|
 |O(3):MLift::[F=0],[Zone=1],[CF=15],[State=Move(Up)],
             [LStop=17.nil]|
 |O(4):Call::[AssignedCall=[3,2.6.nil].[9,1.7.nil].
                          [17,8.23.nil].nil],
            [LCall=[24,6.8.15.19.nil].nil]|

'Main:state' :
 |O(1):MLift::[F=0],[LStop=2.6.nil],[State=Move(Up)],
             [CF=3],[Zone=0]|
 |O(2):MLift::[F=0],[Zone=1],[CF=9],
             [State=Move(Down)],[LStop=9.nil]|
 |O(3):MLift::[F=0],[Zone=1],[CF=16],[State=Move(Up)],
```

```
              [LStop=17.nil]|
 |O(4):Call::[AssignedCall=[9,1.7.nil].
                          [17,8.23.nil].nil],
             [LCall=[24,6.8.15.19.nil].nil]|
...
...

'Main:state' :
 |O(1):MLift::[F=0],[LStop=nil],[State,Move(Down)],
              [CF=6],[Zone=0]|
 |O(2):MLift::[Zone=0],[F=0],[CF=1],[State=Wait],
              [LStop=nil]|
 |O(3):MLift::[Zone=0],[F=0],[CF=8],[State=Wait],
              [LStop=nil]|
 |O(4):Call::[LCall=nil], [AssignedCall=nil]|

'Main:state' :
 |O(1):MLift::[State=Wait],[LStop=nil],[CF=6],
              [F=0],[Zone=0]|
 |O(2):MLift::[Zone=0],[F=0],[CF=1],[State=Wait],
              [LStop=nil]|
 |O(3):MLift::[Zone=0],[F=0],[CF=8],[State=Wait],
              [LStop=nil]|
 |O(4):Call::[LCall=nil],[AssignedCall=nil]|

[] result term:

 |O(1):MLift::[State=Wait],[LStop=nil],[CF=6],
              [F=0],[Zone=0]|
 |O(2):MLift::[Zone=0],[F=0],[CF=1],[State=Wait],
              [LStop=nil]|
 |O(3):MLift::[Zone=0],[F=0],[CF=8],[State=Wait],
              [LStop=nil]|
 |O(4):Call::[LCall=nil],[AssignedCall=nil]|
```

During this execution, we observe the evolution of the set of elevators step by step:

1. At 1st step, three calls are assigned (these three calls are put in the attribute `AssignedCall` of object `O(4)`), one to elevator `O(1)` (the 3rd floor), one to the elevator `O(3)` (the 17th floor) and one to the elevator `O(2)` (the 9th floor). One call has not yet been assigned. This assignment step of calls also selects a direction for each elevator (two go up and one down).

2. The 2nd step does not change a lot of attributes. Each elevator goes on up or down.

3. There is a change at 3rd step because the elevator `O(1)` is already at floor 3 where there is a call. So, the list of assigned calls removes this one and the list of stops of `O(1)` is updated.

4. This process continues for a few steps...

5. The last but one step has no more call. Two elevators are waiting (`O(3)` and `O(2)`) and the `O(1)` elevator is going down at floor 6 without any floor to serve.

6. The last step makes `O(1)` waiting at floor 6.

7. This step cannot be reduced anymore, this is the result term.

## 6 Conclusion

Mixing objects, rules and strategies in a same language gives rise to an increased expressive power. However two main questions remain: efficiency and safety.

Our first experiment obtained by translating the object level into ELAN was a good approach to explore the power of the framework and to understand its semantics in rewriting logic. In order to get an efficient programming language, one needs to go further. A first approach is to translate object programs into an internal term structure directly executable by the ELAN compiler, avoiding in this way to produce new ELAN modules. A second approach is to solve the question of efficiency by the design of a new compiler integrating the concept of objects.

Our example of the multi-elevator controller suggests that interesting properties of the controller should be provable: for instance, one can always reach a state where the lists of calls and stops are empty; or there is no blocking situation. Being backed upon a simple logic like rewriting is a main advantage for proving such properties and designing safe applications. Several works have already been done on Production Rule Systems to verify such systems and to prove their confluence or termination. Let us mention here the COVADIS system (Rousset, 1988) designed to prove the consistency of knowledge-based systems. PREPARE (Zhang and Nguyen, 1994) is also a system able to detect potential errors of rule-based systems. In (Schmolze and Snyder, 1995), Schmolze and Snyder have also defined a tool based on the Knuth-Bendix Completion (Knuth and Bendix, 1970) to test the confluence of Production Rules Systems. In order to prove termination of constraint solver implemented in CHR, Frühwirth has adapted technics usually used in rule-based systems (Frühwirth, 2000). However a challenging question is now to extend these proof techniques to take into account strategies. Indeed a set of rules may be non confluent or non terminating in general but confluent and terminating under a given strategy.

## References


Arnold, K. and Gosling, J. (1996). *The Java programming language.* Addison Wesley.

Birtwistle, G., Dahl, O.-J., Myhrhaug, B., and

Nygaard, K. (1979). *Simula Begin*. Studenlitteratur.

Borovanský, P. (1998). *Le contrôle de la réécriture: étude et implantation d'un formalisme de stratégies*. Thèse de Doctorat d'Université, Université Henri Poincaré – Nancy 1, France. also TR LORIA 98-T-326.

Borovanský, P., Kirchner, C., Kirchner, H., Moreau, P.-E., and Ringeissen, C. (1998). An overview of ELAN. In Kirchner, C. and Kirchner, H., editors, *Proceedings of the 2nd International Workshop on Rewriting Logic and its Applications, WRLA'98*, volume 15, Pont-à-Mousson (France). Electronic Notes in Theoretical Computer Science.

Borovanský, P., Kirchner, C., Kirchner, H., Moreau, P.-E., and Vittek, M. (1996). ELAN: A logical framework based on computational systems. In Meseguer, J., editor, *Proceedings of the first international workshop on rewriting logic*, volume 4 of *Electronic Notes in TCS*, Asilomar (California).

Borovanský, P., Kirchner, C., Kirchner, H., and Ringeissen, C. (1999). Rewriting with strategies in ELAN: a functional semantics. *International Journal of Foundations of Computer Science*.

Frühwirth, T. (2000). Termination of CHR Constraint Solvers. In *this volume*.

Goldberg, A. and Robson, D. (1983). *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley.

Knuth, D. E. and Bendix, P. B. (1970). Simple word problems in universal algebras. In Leech, J., editor, *Computational Problems in Abstract Algebra*, pages 263–297. Pergamon Press, Oxford.

Meyer, B. (1992). *Eiffel: The Language*. Prentice Hall.

Rousset, M.-C. (1988). On the Consistency of Knowledge Bases : The COVADIS System. In *Proceedings of the 8th European Conference on Artificial Intelligence*, pages 79–84.

Schmolze, J. G. and Snyder, W. (1995). A Tool for Testing Confluence of Production Rule Systems. In Ayel, M. and Rousset, M.-C., editors, *Proceedings of the European Symposium on the Validation and Verification of Knowledge-Based Systems. Université de Savoie, Chambéry, France*.

Van den Brand, M., de Jong, H., Klint, P., and Olivier, P. (2000). Efficient annotated terms. *Software-Practice and Experience*, 30:259–291.

Zhang, D. and Nguyen, D. (1994). PREPARE : A Tool for Knowledge Base Verification. *IEEE Transactions on Knowledge and Data Engineering*, 6(6):983–989.