Object Programming in a Rule-Based Language with Strategies

Hubert Dubois, Hélène Kirchner

Object Programming in a Rule-Based Language with Strategies

Hubert Dubois and Hélène Kirchner
LORIA-Université Nancy 2 & LORIA-CNRS
BP 239
54506 Vandœuvre-lès-Nancy Cedex, France
{Hubert.Dubois|Helene.Kirchner}@loria.fr

Abstract. This paper presents a programming framework that combines the concepts of objects, rules and strategies, built as an extension of the rule-based language with strategies ELAN. This extension is implemented in a reflective way in ELAN itself and relies on the same formal semantics, namely the ρ-calculus.

1 Introduction

Object-based languages and rule-based languages have independently emerged as programming paradigms in the eighties. Languages like Ada [Ros92], Smalltalk-80 [GR83], Eiffel [Mey92] or Java [AG96] are well-known and largely used but often lack of semantical basis. Rule-based systems, initially used in the artificial intelligence community, have gained interest with the development of efficient compilers.

The ELAN system [BCD$^+$00] 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 gives 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 [BKKR01] and on rewriting calculus [Cir00]. 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. However, ELAN lack object oriented features, the notion of states, which provides more structuration, and the ability to define structures sharing the same properties.

More recently, the combination of object-based languages and rule-based languages has proved to be quite relevant to formalize and solve advanced industrial problems that require some form of reasoning. Among many other languages, let us mention three of them, more closely related to our approach: CLAIRE [CL96], Oz [HSW95] and Maude [CDE$^+$00].

CLAIRE is combines objects and propagation rules in the logic programming style for problem solving and combinatorial optimization. Single inheritance and full polymorphism are supported. CLAIRE has been realized to deal with applications with complex data structures with the ability to define rules. CLAIRE rules associate a logical condition with an expression composed by one or two objects: when a condition is evaluated into true, the expression is evaluated for the given objects. Conditions are events that denote an evolution of an entity (creation of an object, modification of an attribute, etc...). Rules can be grouped together but control is not explicit.

Oz is a concurrent object oriented constraint language that offers multiple inheritance, higher-order functions and search combinators. The programming language Oz integrates the paradigms of imperative, functional and concurrent constraint programming in a computational framework. The integration of objects into the programming language is interesting because they have defined a small Oz that can support objects for concurrent constraint programming. Propagation rules are also used in the context of constraint solving.

Maude is a language based on rewriting logic, with several extensions, among which Full Maude where object oriented modules can be defined. It offers the possibility to develop concurrent object systems, or *configurations*, where the current state has a multiset structure of objects and messages, and evolves by application of rules that implement message calls.

Compared to these existing languages, extending ELAN with objects provides several advantages. First of all, the fact that ELAN provides a strategy language and strategy constructors to define control on rules, appears as essential for many applications. Second, the main features that characterize object languages (definition of classes composed of attributes and methods; simple inheritance; method call on objects) can be defined in a reflective way, since the extension is defined in ELAN itself. Finally our last, but maybe more important, point is that the extension has a semantics compatible with the rewriting calculus and is achieved by mapping the theory of objects into the ρ-calculus.

In Section 2, we first present the syntax of the object extension of ELAN, including object modules and rules on objects. The encoding of objects and classes into an algebraic theory is presented in Section 3. This provides the basis for the reflective implementation of the extension in ELAN itself. Section 4 first introduces the rewriting calculus on which the semantics of the object extension is defined. The conclusion gives some further perspectives. An extensive example is developed in Appendix.

2 The language extension

Adding object-oriented features to ELAN was motivated by the need of representing structured data and states and to combine them with rewrite rules and strategies describing their evolution. In this section, after a short presentation of the ELAN system, we define the object-oriented extension of ELAN that consists in declaring special modules that we call OModules (OModule for Object Module) where the user can define the classes that he uses. In each module, attributes and methods are defined for each class; the syntax of these object modules is quite similar to object languages like Smalltalk-80 [GR83] or OCaml [RV98].

2.1 ELAN

The starting point of this work was the ELAN system. We briefly present the features of the language that are used in this paper and the reader can refer to [BCD$^+$00] for further details and examples. In order to have some additional informations, the reader can refer to several articles and presentations of the system[1].

The language is close to the algebraic specification formalism with abstract data types defined by operators and rewrite rules, but provides additional specificities that are worth emphasizing. 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. A distinction is made between unlabelled rules for computations, which are required to be confluent and terminating, in order to give a unique result, and labelled rules for deductions, for which no confluence nor termination is required.
- Rules and strategies are first-class objects in the language.
- Application of a rule or a strategy on a term may give zero, one, or several results. This non-determinism related to the production of sets of results is handled by backtracking.

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 this importation may be local (not visible outside the module) or global (visible outside). A module also defines 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, and strategies, also defined by operators and rules.

Predefined modules exist in the ELAN library, such as `bool`, `int`, `ident`, `list[X]`...

Confluent and terminating rules Conditional rewrite rules can be grouped together according to the sort of their left (and right) hand side. For rewrite systems with mutually exclusive conditions in rules, we have confluence and terminating properties. An application of such set of rules on an initial term produces a unique result.

[1] See the Web site of ELAN: *http://elan.loria.fr*

Strategy 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.

- 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 sub-strategies S_i fail.
- **first_one**$(S_1, \ldots, S_n)$ chooses in the list the first strategy S_i that does not fail, and returns its first result. This strategy produces at most one result or fails if all sub-strategies fail.
- The strategy **id** is the identity that does nothing but never fails.
- **fail** is the strategy that always fails.
- **repeat***(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.

But the user may also design his own strategies. 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 class of strategies called elementary strategies. 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 s-trategy constructors and rule labels, but that does not involve S. The application of a strategy S on a term t is denoted $(S)\ t$.

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 with local variables allowing to apply strategies on subterms is allowed in ELAN.

One can also generalize variables to patterns, i.e. terms with variables. We define $\mathcal{T}(\mathcal{F}, \mathcal{X})$ as the set of terms built from a given finite set $\mathcal{F}$ of function symbols and a denumerable set $\mathcal{X}$ of variables and $\mathcal{V}ar(t)$ as the set of variables occurring in a term t. We assume that the reader is familiar with these notations and to get more details, he can refer to the main concepts of general logic [Mes89] and rewriting logic [Mes92].

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

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

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

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

to a subject t, the matching substitution from l to t ($l\sigma = t$) is successively composed with each matching μ_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 normalized 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 is 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.

Associative Commutative functions Associative and commutative (AC for short) 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.

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.

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 flavor of logic programming and allows to program non-deterministic computations. The main originality is surely the capability of strategy programming for expressing the control of programs in a declarative way.

Adding objects oriented features to ELAN amounts to enrich the language with OModules and rules on objects, described below.

2.2 General syntax of OModules

To each class definition corresponds a specific OModule. In order to differentiate these modules from ELAN standard ones, a particular syntax for OModules is introduced, quite similar to object languages like Smalltalk-80 or OCaml [RV98].

In an OModule, the following items are successively defined: the attributes composing each object of this class, the methods associated to the class, the imported modules and the inherited classes.

Attributes. Each object is characterized by its attributes. To each attribute is associated a type (or sort) defining the set of values that it can take. An initial value is also specified for each attribute which is used when a new object of the class is created.

Example 1 *Let us consider the class `Point` defining points in a bidimentionnal space. Two attributes are declared in this class: the first one, called `X`, defines the abscissa of any point and the second one, called `Y`, for the ordinate. This class is declared in the following object module:*

```
class Point
attributes X:int = 0
           Y:int = 0
End
```

Any object of this class is created with an abscissa and an ordinate initialized to 0.

Methods. A method is a function that can be applied to a given object called *target object*. This method can modify the target object. Calling a method `m` on an object `o` is denoted by `o.m`. This is what is usually called *message passing*. We distinguish here two kinds of methods.

First, the generic methods are those that are automatically defined by the system. These methods deal with the modification and the access to the value of an attribute and also with the creation of an object of a given class.

Let us consider a class C where an attribute A of sort t_A is defined. Two methods, $GetA$ and $SetA$, are associated to this class and to this attribute.

- The message $o.GetA$ calls the method $GetA$ on the object o. The result of this method call is of sort t_A and is the value associated to the attribute A of the object o.
- The message $o.SetA(V)$ calls the method $SetA$ on the object o with the parameter V of sort t_A. The result is the object o updated by replacing the value of the attribute A by the given value V.

In each class, a specific method called `new` is defined which constructs a new initial object of the class, whose attributes are initialized with default values. This object has access to all methods defined in the class. Any new object of this class is created as a copy (a clone) of the initial object and can of course be modified later on. This technique is used in actor languages like Scheme [ADH+98] or Common Lisp [Ste90].

The second kind of methods are those defined by the user: the user methods. A user method is given by its name, a list of arguments if necessary, the sort of the result, possibly local variables and its body.

In the method definition, local variables must be declared before being used in the body in local assignments.

A method can have parameters defined by a formal name and a sort. The object designed by the method call is a particular implicit parameter: it can be referred to by the keyword **self** in the body of the method. This parameter does not have to be declared in the list of arguments.

Method bodies are composed of different instructions: the method call, the concatenation of several instructions, boolean tests and local assignments. These instructions are illustrated in the next example.

Example 2 *Let us consider the class `PointTranslation` with two attributes `X` and `Y` of sort `int` initialized to `0` and which defines two methods: a first one, `TranslateX`, modifies the value of the attribute `X` by adding the value of a parameter `N` to the old value assigned to `X`. The second one, `Translate`, calls the translation over `X` only if a condition on the values of `X` and `Y` is checked. If this is the case, a value for the local variable `N` is computed to allow the method call of `Translate` parameterized by `N`.*

```
class PointTranslation
attributes X:int = 0
           Y:int = 0
method TranslateX(N:int) for PointTranslation
          <self.SetX(self.GetX + N)>
method Translate for PointTranslation N:int
          <if self.GetX > self.GetY ;
           N := self.GetX - self.GetY ;
           self.TranslateX(N)>
End
```

Importation of modules. Each object module can import others modules which are not object modules but standard ELAN modules where sorts, operators, rules and strategies are defined. An object module may use a library of standard ELAN modules, but is not allowed to import another object module. In this way, these importations are different from the inheritance mechanism of object languages presented below.

Example 3 *Let us consider the class Point defined in the example 1. Now, we enrich it in order to define colored points. The colors of the points are defined in a standard* ELAN *module that is called `color.eln` in which the sort `color` is defined as the sort that enumerates all the possible color values. Let us consider that `Black` is one of these values.*

We introduce the new class of colored points ColoredPoint, where the attribute denoting the color of each point is the attribute `Color`, of sort `color` *initialized with the value* `Black`*, we define the object module ColoredPoint as follows:*

```
class ColoredPoint
imports color
attributes X:int = 0
           Y:int = 0
           Color:color = Black
End
```

Inheritance. An object module can inherit attributes and methods from another class by using the keyword **from** followed by a class name. The inheritance mechanism allows a given class `B`, inheriting a class `A`, to specialize the inherited class: attributes and methods defined in `A` are available, but new attributes and new methods can then be defined too. Methods can also be overridden in order to specialize them. This kind of inheritance is called simple inheritance.

The inherited methods are overloaded with the definition of methods with the same name but their sorts are different according to the associated class. The overloading of methods can also be used to redefine a method. The sort and the body of the method are then different between the two classes. This feature also holds for attributes that can be overloaded by giving another initial value.

Example 4 *Let us consider the example of the class Point defining basic points given in Example 1 and the class ColoredPoint defining colored points given in the Example 3. Instead of completely redefining the class ColoredPoint as before, inheritance can be used to simply define the ColoredPoint class from the Point class as follows:*

```
class ColoredPoint
imports color
from Point
attributes Color:color = Black
End
```

The attributes `X` *and* `Y` *from class Point are inherited, as well as the particular methods* `GetX`*,* `GetY`*,* `SetX` *and* `SetY` *associated to the Point class. Thus, we only have to define the attribute* `Color` *while the associated methods* `GetColor` *and* `SetColor` *are then automatically associated.*

The simple inheritance that is defined here can be compared to simple inheritance in Java but is less powerful than in Smalltalk-80 where inheritance is multiple, which means that a class can inherit several other classes.

The complete syntax of OModules and more examples can be found in [Dub01].

2.3 Rules on objects

An object base represents the set of all objects that live at a given time in the system. Rules are defined to delete, modify or add objects in this object base. An informal presentation is given here.

Rules are of the form:

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

where $O_1, \ldots, O_k, O'_1, \ldots, O'_m$ are objects, t is a boolean term and l a local assignment, that both may involve objects and method calls. This rule, as standard ELAN rules previously presented, can be labelled. The O_i objects of the left-hand side represent the matching conditions of this rule. The rule is applied only if these objects can be found in the object base. Each rule mentions the relevant information on the object base; the context is omitted. The order of objects O_i and O'_j in both sides is not relevant since an AC operator is used for the objects base construction. Each O_i object of the left-hand side has one of two possible forms: the first one, $O_i : ClassName_i :: [Att_1(Value_1)\ ,\ \ldots\ ,\ Att_n(Value_n)]$, corresponds to an object of a given class $ClassName_i$ where few attributes Att_i are specified and the second form, $O_i : ClassName_i$, corresponds to an object for which only the class $ClassName_i$ is specified.

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.

Including objects in rules controlled by strategies is now possible in the defined formalism of rules working on an object base. Planification or scheduling problems are easily expressed as shown in [DK00b] where the formalism of rules presented in this paper is enriched in order to also control a constraint base. Strategies are used in both cases to control the application of the rules. Other examples of applications can be found in [Dub01].

3 An algebraic encoding of objects

Our purpose now is to show how the definition of classes in the object modules can be implemented in a first-order algebraic language such as ELAN. We choose here an approach where objects and classes are represented as objects like in the class-based language Smalltalk-80 where the unique entity is the object. This implementation has been guided by a few choices.

There is a distinction between attributes and methods. In many applications, using objects essentially consists in reading or modifying the values of attributes. Thus, we have to define a structure where the values of attributes are quickly accessible. The difference between attributes and methods is a good way to distinguish what represents the state of an object at a given time (the attributes and their associated values) and the functions that can be applied to this object (the methods). However, we do not want to separate an object with its attributes from the set of methods that can be applied to the object. We thus have to define a structure where each object includes the methods associated to it.

The application of a method is defined with rewrite rules. This imposes some (quite reasonable) restrictions on the definition of methods and is compatible with the distinction between attributes and methods in the following way: to an attribute is associated a value which can be modified during the evaluation, but no rewrite rule. An attribute is mutable by rewriting, and complex expressions may be considered as values of attributes. On the contrary, methods are defined by rewrite rules and are non-mutable, i.e. once rewrite rules associated to methods have been defined, one cannot delete or even change them. Thus, an object is composed of mutable attributes, whose values can be changed during evaluation, and of references to non-mutable methods. The reference to any method can be hidden or revealed during the execution and thus, the ability for an object to execute a method can change.

3.1 A representation based on operators and rules

Each object composed of attributes and methods which the object has access to is represented by a term; methods are represented by functions defined by rewriting rules.

- In order to represent the attributes and their corresponding values, the object structure involves a set of pairs *(attribute,value)*. Each attribute is denoted by a constant and each value by a term with the same sort.
- The object structure also involves a set of constants denoting methods. To each of them, an operator is associated that has the same name. To each method body corresponds a rewrite rule right-hand side, with, possibly, local assignments and boolean tests.

This object representation is schematized in Figure 1.

Definition of operators and rules associated to the object representation are detailed in Section 3.2 and in Section 3.3. Operators and rules associated to user methods are detailed in Section 3.4.

3.2 Operators associated to the representation

The signature of operators that are defined to build the representation of objects is as follows:

$$\begin{array}{lllll}
[\_] & : Methods & \mapsto Object & \\
\_,\_ & : Methods \times Methods & \mapsto Methods & (AC) \\
\_ & : Method & \mapsto Methods & \\
\_(\_) & : MName \times MBody & \mapsto Method & \\
\_ & : MName & \mapsto Method &
\end{array}$$

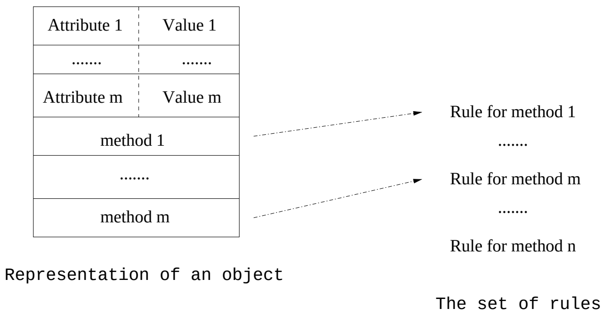

Fig. 1. Object representation

The notation used is the following: each operator takes arguments that are denoted by _. The sort of these arguments are given in the left part of the profile while the right part is the sort of the result. An operator is mixfix, that is to say that an argument can appear anywhere in the operator definition. For instance, `_(_)` has two arguments of respective sorts $MName$ and $MBody$ and gives a result of sort $Method$. This is a mixfix operators that builds the association *(attribute,value)*: the name of the attribute (of sort $MName$) appears in the first place and the corresponding value (of sort $MBody$) between the brackets.

`[_]` is the constructor of objects. Each object o has the form $[LM]$ where LM, of sort $Methods$ is a set composed of pairs *attribute(value)* and of references to methods. This set is built with the operator `_,_` which is associative and commutative; this is indicated by annotation (AC). Each element, of sort $Method$, is thus either a pair *(attribute,value)*, or a reference to a method (the last _ operator) also of sort $MName$.

To these declarations and definitions, we also add rules that correspond to the manipulation of objects: to add an attribute, to modify its value, to access a method and to create a new object. These rules are defined in Section 3.3. But, before defining the rules, we have to present the corresponding operators:

$$\begin{array}{lll} add(\_,\_) & : Object \times Method & \mapsto Object \\ kill(\_,\_) & : Object \times MName & \mapsto Object \\ access(\_,\_) & : Object \times MName & \mapsto MBody \\ new(\_) & : Object & \mapsto Object \end{array}$$

These four basic operators are defined and used to construct and decompose objects: the `add(_,_)` operator is used to add a new element to the set of attributes and methods; the `kill(_,_)` operator is used to remove an element given in the parameters from the set of pairs and references that compose the object; the `access(_,_)` operator is used to have access to the value associated to an attribute given as parameter; the last `new` operator is used to create a new object from the object representing the class.

In the set of methods, the two particular methods `Get` and `Set` take as arguments the attribute they deal with.

$$\begin{aligned} Get(\_,\_) &: Object \times MName \mapsto MBody \\ Set(\_,\_,\_) &: Object \times MName \times MBody \mapsto Object \end{aligned}$$

3.3 Rules associated to the representation

We now define a system $\mathcal{R}$ of rewriting rules that are used to evaluate objects. This system is based on the above representation of objects and inspired from the object ρ-calculus [CKL01]. The whole rewrite system $\mathcal{R}$ can be found in Figure 2. $\mathcal{R}$ is composed of rules related to the object definition and manipulation. We have proved in [Dub01] that the system $\mathcal{R}$ is confluent and terminating.

Add a component	$add([LM], me) \to_{\mathcal{R}} [LM, me]$
Remove a component-1	$kill([M(B), LM], M) \to_{\mathcal{R}} [LM]$
Remove a component-2	$kill([M, LM], M) \to_{\mathcal{R}} [LM]$
Access to an attribute value	$access([M(B), LM], M) \to_{\mathcal{R}} B$
Access to a value by `Get`	$Get(o, M) \to_{\mathcal{R}} access(o, M)$
Modification of a value by `Set`	$Set(o, M, B) \to_{\mathcal{R}} add(kill(o, M), M(B))$
Creation of a new object	$new(o) \to_{\mathcal{R}} [a_1(vi_1), \ldots, a_n(vi_n), m_1, \ldots, m_m]$
The operator `Geta`	$Get_a(o) \to_{\mathcal{R}} Get(o, a)$
The operator `Seta`	$Set_a(o, v) \to_{\mathcal{R}} Set(o, a, v)$
Method call for `Geta`	$[LM, Get_a].Get_a \to_{\mathcal{R}} Get_a([LM, Get_a])$
Method call for `Seta`	$[LM, Set_a].Set_a(v) \to_{\mathcal{R}} Set_a([LM, Set_a], v)$
Method call for `new`	$[LM, new].new \to_{\mathcal{R}} new([LM, new])$
Method call for a method `m`	$[LM, m].m(p_1, \ldots, p_m) \to_{\mathcal{R}} m([LM, m], p_1, \ldots, p_m)$

Fig. 2. The system $\mathcal{R}$

3.4 Operators and rules associated to user's methods

For each user methods, we associate:

- a constant of sort $MName$ representing the name of the method;
- an operator declaration which profile is based on the profile of the user's method: to each parameter of the method corresponds an argument of the operator. We also add to the corresponding operator a first argument which represents the object itself (the **self**);
- a set of rules defining this operator.

Let us now present how the rules associated to user's methods are built. This is performed by defining a transformation mechanism that deduces rewrite rules for each user method. The operator $build-rule(\_)$ (cf. Figure 3) takes as argument the definition of a method in the formalism described in Section 2.2 and returns the associated rewrite rule in the ELAN syntax.

$$
\left\{
\begin{array}{lll}
build-rule(\textbf{ method } name\ (args) \textbf{ for } t\ vars\ body) = & & \\
 & \textbf{rules for } t & \\
 & \quad S\ :\ class\_name; & \\
 & \quad var-decl(vars, args, body) & \\
 & \quad [\,]\ name(S, get-args(args))\ \ {=}{>}\ \ build-rhs(body, list\_vars) \textbf{ end} & \\
 & \textbf{end} & \\
build-rule(\textbf{ method } name\ (args) \textbf{ for } t\ body) & = & \\
 & \textbf{rules for } t & \\
 & \quad S\ :\ class\_name; & \\
 & \quad var-decl(args, body) & \\
 & \quad [\,]\ name(S, get-args(args))\ \ {=}{>}\ \ build-rhs(body, list\_vars) \textbf{ end} & \\
 & \textbf{end} & \\
build-rule(\textbf{ method } name \textbf{ for } t\ vars\ body) & = & \\
 & \textbf{rules for } t & \\
 & \quad S\ :\ class\_name; & \\
 & \quad var-decl(vars, body) & \\
 & \quad [\,]\ name(S)\ \ {=}{>}\ \ build-rhs(body, list\_vars) \textbf{ end} & \\
 & \textbf{end} & \\
build-rule(\textbf{ method } name \textbf{ for } t\ body) & = & \\
 & \textbf{rules for } t & \\
 & \quad S\ :\ class\_name; & \\
 & \quad var-decl(body) & \\
 & \quad [\,]\ name(S)\ \ {=}{>}\ \ build-rhs(body, list\_vars) \textbf{ end} & \\
 & \textbf{end} &
\end{array}
\right.
$$

Fig. 3. Definition of function $build-rule$

In the definition of $build-rule$, we handle the four possibilities that can appear and which depend on having or not arguments ($args$) and local variable declarations ($vars$). The rules are built in the ELAN syntax.

The operator $build-rhs(\_,\_)$ takes a body and a list of variables which is built step by step. Each new variable added to this list corresponds to a local variable used when an object is modified. This list memorizes this information. When initialized, this variable list is only composed of the variable S.

Example 5 *Let us consider the Translate method presented in Exemple 2. The rule corresponding to this method is defined by build − rule and the result is:*

```
rules for PointTranslation
 N  : int;
 S  : PointTranslation;
 O1 : PointTranslation;
[] Translate(S) => O1
      if GetX(S) > GetY(S)
      where N := () GetX(S) - GetY(S)
```

```
    where O1 := () TranslateX(S,N)
```

More details can be found in [Dub01].

The system $\mathcal{R}$ presented in Section 3.3 is thus enriched with the rules defining the user methods to give a set $\mathcal{R}'$. For the time being, the system does not check that $\mathcal{R}'$ is confluent and terminating and this is under the user's responsibility. Indeed, we aimed at a complete proof environment that would automatically check these properties.

4 An operational semantics based on the ρ-calculus

Our goal is now to give a formal semantics to this object-oriented extension of ELAN. Several calculus as the *Object Lambda Calculus* defined by K. Fisher, F. Honsell and J.C. Mitchell [FHM94] and the *Object Calculus* of M. Abadi and L. Cardelli [AC96] are candidates to provide object languages with a formal semantics. Our choice is to base the semantics on another calculus called the Rewriting Calculus, or ρ-calculus defined by H. Cirstea and C. Kirchner [CK99] that encompasses in particular both λ-calculus and term rewriting. In this calculus, terms, rewriting rules and application of a rule on a term can be represented.

This choice was done for two reasons: first, the ρ-calculus already provides an operational semantics for ELAN and second, it is general enough to represent both the *Object Lambda Calculus* and the *Object Calculus* as shown in [CKL01], where an extension of the ρ-calculus, called the object ρ-calculus has been defined.

In order to prove that the ρ-calculus gives an operational semantics to the object extension of ELAN, we establish a close correspondence between reduction of an object term in the algebraic theory of objects given in Section 3 and reduction of the corresponding ρ-term in the ρ-calculus.

In the algebraic theory of objects, an object is represented as a term of sort `Object`. We have proved that the set of rules $\mathcal{R}$ is terminating and confluent and we can consider a confluent and terminating extended set $\mathcal{R}'$ with rewriting rules corresponding to user-defined methods. So each term t of sort `Object` has a normal form denoted by $NF_{\mathcal{R}'}(t)$, or $NF(t)$ for short.

In Figure 4, we illustrate that each reduction (noted $\dashrightarrow$) of a term t of sort `Object` to its normal form $NF(t)$ corresponds to a reduction (noted $\rightarrow$) in the ρ-calculus from a ρ-term t_0 to another one t'_0, where t_0 and t'_0 are the respective translations of t and $NF(t)$ considering a translation τ. Definition of τ, results and proofs can be found in [Dub01].

Once object terms have been translated to ρ-terms, a data base of objects is translated to a ρ-term built with an AC-operator on top (a set of ρ-terms). Rules on objects also correspond to ρ-terms [CK99] and application of these rules to the data base is application of ρ-calculus.

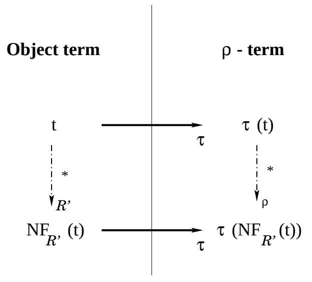

Fig. 4. Correspondence between object rewriting and ρ-term evaluation

5 Conclusion

The purpose of this paper was to show that objects, rewriting rules and strategies can be integrated in a same formalism with an operational semantics defined by the ρ-calculus.

We have proposed a language to define objects in the "programming by rewrite rules and strategies" paradigm offered by the ELAN system. This consists in adding OModules to the standard modules where operators, rules and strategies are defined in ELAN; in OModules, classes with attributes and methods are declared. Mixing OModules with standard modules allows the developer to define applications where special rules that manage objects can be defined. Furthermore, we have shown that objects can also be represented in the ρ-calculus that already gives an operational semantics to ELAN. The purpose of this work was to provide object features in ELAN in a semantically coherent way, without pretending to design a new powerful object oriented language. On the other hand, thanks to the operational semantics based on the rewriting calculus, one can develop verification tools that often lack in classical object oriented languages.

Developing objects in a rewriting context is also useful for rewriting itself. Indeed, when considering only a part of the object base in an object rule, this allows a better structure of the knowledge and this also allows a kind of global variable represented by the object base. This is very useful in a rewriting context where the possibility to use global variables often lacks.

In [DK00b], a formalism where rules may also be extended with constraints is presented. This leads to a general framework where rewriting rules and strategies can manage simultaneously an object base and a constraint base. As the constraint solver used in the applications definition is also based on the rewrite system ELAN, and, thus, on the ρ-calculus semantics, this complete framework is also based on the ρ-calculus semantics [Dub01].

Some applications of ELAN extended with objects have been developed. A first one consists in defining a multi-elevator controller [DK00a]. A second one consists in defining a print controller [DK00b]; in this case, rules are extended with objects and also with constraints. Planification and scheduling problems are then very easily defined in this formalism.

In such applications, strategies to control the applications of rules on objects have been proved useful. A further interesting direction is to use strategies to define methods, especially non deterministic ones, inside OModules. Although this is not a problem at the semantical level, this would need to implement in ELAN an explicit application operator.

Adding new components to the standard ELAN system such that OModules and new kind of rewrite rules was made possible by using a transformation process of OModules into ELAN modules, which relies on the algebraic theory of objects presented in this paper. The extended language has been prototyped in ELAN in this way, and more details can be found in [DK00a,Dub01]. Although this first experiment was a good approach to explore the power of the framework, in order to get an efficient programming language, one needs to go further. A more promising approach, currently explored, 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.

References

[Abr96] J.-R. Abrial. *The B-Book: Assigning Programs to Meanings.* Cambridge University Press, 1996.

[AC96] M. Abadi and L. Cardelli. *A Theory of Objects.* Springer-Verlag, 1996.

[ADH$^+$98] H. Abelson, R. K. Dybvig, C. T. Haynes, G. J. Rozas, N. I. Adams IV, D. P. Friedman, E. Kohlbecker, G. L. Steele Jr., D. H. Bartley, R. Halstead, D. Oxley, G. J. Sussman, G. Brooks, C. Hanson, K. M. Pitman, and M. Wand. Revised^5 report on the algorithmic language Scheme. *Higher-Order and Symbolic Computation*, 11(1):7–105, August 1998.

[AG96] K. Arnold and J. Gosling. *The Java programming language.* Addison Wesley, 1996.

[Bar85] H. Barringer. Up and down the temporal way. Technical Report UMCS–85–9–3, Department of Computer Science, Manchester University, Oxford Rd., Manchester M13 9PL, UK, 1985.

[BCD$^+$00] P. Borovanský, H. Cirstea, H. Dubois, C. Kirchner, H. Kirchner, P.-E. Moreau, C. Ringeissen, and M. Vittek. ELAN *V 3.4 User Manual.* LORIA, Nancy (France), fourth edition, January 2000.

[BKKR01] P. Borovanský, C. Kirchner, H. Kirchner, and C. Ringeissen. Rewriting with strategies in ELAN: a functional semantics. *International Journal of Foundations of Computer Science*, 2001.

[CDE$^+$00] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martì-Oliet, J. Meseguer, and J.F. Quesada. Towards Maude 2.0. In K. Futatsugi, editor, *WRLA2000, the 3rd International Workshop on Rewriting Logic and its Applications, September 2000, Kanazawa, Japon.* Electronic Notes in Theoretical Computer Science, 2000.

[Cir00] H. Cirstea. *Le Rho Calcul: Fondements et Applications.* PhD thesis, Université Henri Poincaré - Nancy 1, 2000.

[CK99] H. Cirstea and C. Kirchner. Combining higher-order and first-order computation using ρ-calculus: Towards a semantics of ELAN. In D. Gabbay and

M. de Rijke, editors, *Frontiers of Combining Systems 2*, Research Studies, ISBN 0863802524, pages 95–120. Wiley, 1999.

[CKL01] H. Cirstea, C. Kirchner, and L. Liquori. Matching Power. In A. Middeldorp, editor, *Proceedings 12th Conference on Rewriting Techniques and Applications, RTA 2001, Utrecht, The Netherlands*, volume 2051 of *Lecture Notes in Computer Science*, pages 77–92. Springer-Verlag, 2001.

[CL96] Y. Caseau and F. Laburthe. CLAIRE: Combining objects and rules for problem solving. In *Proceedings of the JICSLP'96 workshop on multi-paradigm logic programming,TU Berlin, Germany*, 1996.

[DK00a] H. Dubois and H. Kirchner. Objects, rules and strategies in ELAN. In *Proceedings of the second AMAST workshop on Algebraic Methods in Language Processing, Iowa City, Iowa, USA*, May 2000.

[DK00b] H. Dubois and H. Kirchner. Rule Based Programming with Constraints and Strategies. In K.R. Apt, A.C. A. C. Kakas, E. Monfroy, and F. Rossi, editors, *New Trends in Constraints, Papers from the Joint ERCIM/Compulog-Net Workshop, Cyprus, October 25-27, 1999*, volume 1865 of *Lecture Notes in Artificial Intelligence*, pages 274–297. Springer-Verlag, 2000.

[Dub01] H. Dubois. *Systèmes de Règles de Production et Calcul de Réécriture.* PhD thesis, Université Henri Poincaré - Nancy 1, 2001.

[FHM94] K. Fisher, F. Honsell, and J. C. Mitchell. A Lambda Calculus of Objects and Method Specializatio n. *Nordic Journal of Computing*, 1(1):3–37, 1994.

[GR83] A. Goldberg and D. Robson. *Smalltalk-80: The Language and Its Implementation.* Addison-Wesley, 1983.

[HSW95] M. Henz, G. Smolka, and J. Wurtz. Object-oriented concurrent constraint programming in oz. In V. Saraswat and P. van Hentenryck, editors, *Principles and Practice of Constraint Programming.*, pages 27–48. MIT Press, 1995.

[Mes89] J. Meseguer. General logics. In H.-D. Ebbinghaus et al., editor, *Logic Colloquium'87*, pages 275–329. Elsevier Science Publishers B. V. (North-Holland), 1989.

[Mes92] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.

[Mey92] B. Meyer. *Eiffel: The Language.* Prentice Hall, 1992.

[Ros92] J.-P. Rosen. What Orientation Should Ada Objects Take? *Communications of the ACM*, 35(11):71–76, 1992.

[RV98] D. Rémy and J. Vouillon. Objective ML: An effective object-oriented extension of ML. *Theory and Practice of Object Systems*, 4(1):27–52, 1998.

[Ste90] G. Steele, Jr. *Common* LISP: *The Language.* Digital Press, Bedford, Massachusetts, 2nd edition, 1990.

[ZM93] Y. Zhang and Alan K. Mackworth. Design and analysis of embedded real-time systems: An elevator case study. Technical Report TR-93-04, Department of Computer Science, University of British Columbia, February 1993.

A An example

In order to illustrate now the use of objects in the extended language, let us consider a program whose purpose is to automate and control an elevator system for buildings with multiple elevators.

Several implementations of a multi-elevator controller exist in the literature, for instance based on constraint nets [ZM93], on temporal logic [Bar85] or on the *Abstract State Machine* [Abr96]. Although not original, this example nicely illustrates the use of ours formalism based on rules, objects and strategies. Compared to other approaches, ours is uniform, the rules are clear and the ability for the user to develop easily strategies to express control is very appealing. Moreover, verification techniques available in rule formalism can be adapted to prove properties such as termination, confluence, completeness of the specification.

To formalize the multi-elevator controller, we first define two classes: a class `MLift` for elevators and a class `Call` for the controller. Then, we present the rules defined to design the multi-elevator controller and, finally, the strategies before a short presentation of an execution.

A.1 The class `MLift`

This class describes elevators. Each elevator is an object of this class, characterized by:

– its current floor denoted by the attribute `CF` -*current floor*- represented by an integer,
– its state: is it going up?, down?, or is it waiting for a call? This is defined by the attribute `State` of sort `LiftState`,
– its list of floors where it has to stop with the attribute `LStop` which is a list of integers.

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`. This is defined in a standard ELAN module by:

```
operators global
    Up        : Sense;
    Down      : Sense;
    Move(@)   : (Sense) LiftState;
    Wait      : LiftState;
end
```

Three other attributes are also defined:

– `Zone` that indicates the zone where this elevator is. Indeed, each building is divided in several zones, each one composed by consecutive floors, and there is as many zones as elevators. For instance, considering a building with 4 elevators and with 27 floors, four zones will be considered: the first one from floor 0 to 6, then, a second one (floor 7 to 13), a third one (floor 14 to 20) and a last fourth one form floor 21 to 27. This attribute is useful if we want to guarantee that when dividing the number of floors into a number of zones,

each zone does not have more than two elevators working at any moment. This guarantees a better quality of service for the access to an elevator in the building.
– F (standing for Flag) whose value is either 0 or 1 indicates that an elevator is performing an instruction (F to 1) or waiting for a new instruction (F to 0).
– I, standing for Interruption, is an integer which can take value in $\{0, 1\}$. If I$= 0$, then the elevator has no interruption and is available; if I$= 1$, then, the elevator is out of service.

Several methods are defined for the class MLift:

– a method WhichSense(_) takes an integer representing the new floor that the elevator has to reach from its current one. It returns the direction that the elevator has to adopt: either it will go up or down.
– the method UpdateZone computes, after the elevator has moved, in which zone it is.
– a method AddLStop(_) takes a list of integers L representing a list of different floors and returns the object representing the elevator whose attribute LStop (list of floors) has been updated with L. The new list LStop is sorted.
– the last method, RemoveLStop(_), deletes from the list of stops LStop the floor given as parameter of this method.

These definitions and declarations of methods and attributes are grouped together in the definition of the OModule for the class MLift:

```
class MLift

imports ToolsMLift

attributes CF:int = 0
           State:LiftState = Wait
           LStop:list[int] = nil
           Zone:int = 0
           F:int = 0
           I:int = 0

method WhichSense(N:int) for MLift
 S : Sense;
 <S:=ChooseSense(self.GetCF,N) ; self.SetState(Move(S))>

method UpdateZone for MLift
 <self.SetZone(NewZone(self.GetCF))>

method AddLStop(L:list[int]) for MLift
 <self.SetLStop(AddAndSort(self.GetLStop,L))>

method RemoveLStop(N:int) for MLift
 <self.SetLStop(RemoveList(self.GetLStop,N))>
End
```

A.2 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 formalized by an attribute `LCall` composed of a list of integers: the requested floor. These are floors that have to be served to load people.

To distinguish calls that are processed from those that are waiting, a second attribute `AssignedCall` is composed of calls that have been assigned to an elevator and which are to be processed.

Different methods are defined in the object module describing the class `Call`:

– a method `AddAssignedCall(_)` takes an integer that represents a floor and adds it in the list of calls that are processed.
– a method `RemoveAssignedCall(_)` takes an integer that represents a floor and removes it from the list of calls that are currently processed.
– a last method called `RemoveLCall(_)` takes an integer that represents a floor and removes it from the list of calls that are waiting to be processed.

The class `Call` is defined in the following OModule:

```
class Call

imports Tools

attributes LCall:list[int] = nil
           AssignedCall:list[int] = nil

method AddAssignedCall(N:int) for Call
 <self.SetAssignedCall(AddList(self.GetAssignedCall,N))>

method RemoveAssignedCall(N:int) for Call
 <self.SetAssignedCall(RemoveList(self.GetAssignedCall,N))>

method RemoveLCall(N:int) for Call
 <self.SetLCall(RemoveList(self.GetLCall,N))>

End
```

A.3 The rules

The rules that define the actions on elevators can now be 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).UpdateZone(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).UpdateZone(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)]
      =>
    O1(State<-Move(Down),F<-1)
      if O1.CF > Max(O1.LStop) or O1.CF == MaxLevel

[ChangeToUp]
    O1:MLift::[State(Move(Down)) , F(0) , I(0)]
      =>
    O1(State<-Move(Up),F<-1)
      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::[F(0) , I(0)]
        =>
    O1.RemoveLStop(O1.CF)(F<-1)
        if O1.State != Wait
        if in(O1.CF,O1.LStop)
```

```
[OpenDoorsCall]
    O1:MLift::[F(0) , I(0)]
    O2:Call
       =>
    O1.AddLStop(L1)(F<-1)
    O2.RemoveLCall(O1.CF)
       if O1.State != Wait
       if in(O1.CF,O2.LCall)
       where L1 := () ObtainNewStops(O1.CF)
```

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::[F(0) , I(0)]
     O2:Call
       =>
     O1.AddLStop(L1).RemoveLStop(O1.CF)(F<-1)
     O2.RemoveLCall(O1.CF)
       if O1.State != Wait
       if in(O1.CF,O1.LStop)
       if in(O1.CF,O2.LCall)
       where L1 := () ObtainNewStops(O1.CF)
```

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, the two objects are updated 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 go for the selected elevator.

```
[AssignACall]
    O1:MLift::[State(Wait) , F(0) , I(0)]
    O2:Call
       =>
    O1.WhichSense(NextFloor).AddLStop(NextFloor.nil)(F<-1)
    O2.AddAssignedCall(NextFloor).RemoveLCall(NextFloor)
     if O2.LCall != nil
     where NextFloor := () ChooseNextFloor(O1.CF,O2.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. It also uses the function `ObtainNewStops` that asks people inside the elevator which floors they want to go to.

```
[OpenDoorsAssignedCall]
    O1:MLift::[F(0) , I(0)]
    O2:Call
       =>
    O1.AddLStop(L1).RemoveLStop(O1.CF)(F<-1)
    O2.RemoveAssignedCall(O1.CF)
       if O1.State != Wait
       if in(O1.CF,O2.AssignedCall)
       where L1 := () ObtainNewStops(O1.CF)
```

A condition to assign a call to an elevator is that at least one elevator has 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::[F(0) , I(0)]
          =>
        O1(State<-Wait)
          if O1.State != Wait
          if O1.LStop = nil
```

A.4 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 checks 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 `ALLLIFTS`:

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

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 inside an elevator, 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
```

The labelled rule `Main` is defined as:

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

A.5 The execution

Let us consider an initial situation described as:

```
O(1):MLift::[CF(14) , State(Wait) , Zone(1) , LStop(nil) , F(0) , I(0)]
O(2):MLift::[CF(11) , State(Wait) , Zone(1) , LStop(nil) , F(0) , I(0)]
O(3):MLift::[CF(2)  , State(Wait) , Zone(0) , LStop(nil) , F(0) , I(0)]
O(4):Call::[AssignedCall(nil) , LCall(3.9.17.24.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.

Applying the `MAIN` strategy to this initial term leads to the following execution:

```
O(1):MLift::[CF(14) , State(Move(Up))   , Zone(1) , LStop(17.nil) , F(0) , I(0)]
O(2):MLift::[CF(11) , State(Move(Down)) , Zone(1) , LStop(9.nil)  , F(0) , I(0)]
O(3):MLift::[CF(2)  , State(Move(Up))   , Zone(0) , LStop(3.nil)  , F(0) , I(0)]
O(4):Call::[AssignedCall(3.9.17.nil) , LCall(24.nil)]

O(1):MLift::[CF(15) , State(Move(Up))   , Zone(1) , LStop(17.nil) , F(0) , I(0)]
O(2):MLift::[CF(10) , State(Move(Down)) , Zone(1) , LStop(9.nil)  , F(0) , I(0)]
O(3):MLift::[CF(3)  , State(Move(Up))   , Zone(0) , LStop(3.nil)  , F(0) , I(0)]
O(4):Call::[AssignedCall(3.9.17.nil) , LCall(24.nil)]

An elevator is stopped at level 3, please enter the desired stops
as a list of sorted integers separated by . and terminated by end:
```

After the user has entered different stops that will be considered by the elevator controller, several execution steps occur and lead, finally, to the two last steps below:

```
...
O(1):MLift::[CF(17) , State(Wait)     , Zone(1) , LStop(nil) , F(0) , I(0)]
O(2):MLift::[CF(10) , State(Wait)     , Zone(1) , LStop(nil) , F(0) , I(0)]
O(3):MLift::[CF(24) , State(Move(Up)) , Zone(2) , LStop(nil) , F(0) , I(0)]
O(4):Call::[AssignedCall(nil) , LCall(nil)]

O(1):MLift::[CF(24) , State(Wait) , Zone(2) , LStop(nil) , F(0) , I(0)]
O(2):MLift::[CF(17) , State(Wait) , Zone(1) , LStop(nil) , F(0) , I(0)]
O(3):MLift::[CF(10) , State(Wait) , Zone(1) , LStop(nil) , F(0) , I(0)]
O(4):Call::[AssignedCall(nil) , LCall(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 17th floor), one to the elevator `O(2)` (the 9th floor) and one to the elevator `O(3)` (the 3rd 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. We can notice that one elevator, the one called `O(3)`, has reached the requested 3rd floor. Then, the operation `ObtainNewStops` can be performed and it consists in asking user new stops for this elevator. This uses the inputs/outputs of the ELAN system.
3. This process continues for a few steps...
4. The last but one step has no more call. Two elevators are waiting (`O(1)` and `O(2)`) and the `O(3)` elevator is going down, it is at floor 24 without any floor to serve.
5. The last step makes the previous elevator waiting at floor 24. This step cannot be reduced anymore, this is the result term.