Rules, strategies and objects in ELAN

Hubert Dubois, Hélène Kirchner

Rules, strategies and objects in ELAN

Hubert Dubois and Hélène Kirchner
LORIA-UHP & LORIA-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 the extension of the language with objects. The extension with objects is defined as a specific instance of the rewriting calculus, also called ρ-calculus. This leads to an expressive programming framework that combines the concepts of objects, rules and strategies which has been prototyped in ELAN.

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 only a few. Rule-based systems are largely used in the artificial intelligence community, while 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 others introduced for evaluation of programming languages. In most programming languages, the evaluation strategy is fixed which makes the evaluation process easier to implement but 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. These ideas are studied, implemented and experimented in ELAN.

The goal of this paper is to present an extension of the ELAN language with objects and some object-oriented features. 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 and the same methods. This object-oriented language relies on the

rewriting calculus developed in [CK99a,CKL00] on which ELAN is also backed. The ELAN extension with objects is defined as a specific instance of the calculus and as an algebraic rewrite theory, precisely related to the rewriting calculus. From the operational point of view, this language extension has been prototyped using also an algebraic approach. The object modules (OModules for short) are compiled into ELAN modules, which makes them executable with the ELAN rewrite engine. The compilation process is itself written in ELAN. Finally, rules and strategies are well suited to describe the dynamic and concurrent evolution of a date base of objects. 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 proposes an extension of ELAN with classes, objects and methods. Section 4 presents the ρ-calculus and its specific instance for ELAN with objects. Section 5 shows how to use ELAN in order to transform an object-oriented program into a basic ELAN program that can be executed. Section 6 shows how to incorporate objects, rules and strategies in the same environment.

2 ELAN

The ELAN system [BKK+96,BKK+98] 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]. 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 a 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-deterministic computations. The main originality is surely the capability of strategy programming for expressing the control of programs in a declarative way. We briefly present the features of the language that are used in this paper and the reader can refer to [BCD+99] for further details and examples.

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 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.

In order to progressively introduce all these ingredients, let us first consider a simple module 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.

```
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
```

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]`.

2.2 Confluent and terminating rules

The previous module illustrates how conditional rewrite rules are grouped together according to the sort of their left (and 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 previous example. First, to achieve this 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 `enumStrat` 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 `enum(i,j)` concatenated with the rule `[final]` which returns `i`. The whole program is defined as follows:

```
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
  enumStrat : <int->int>          ; end

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

The application of the strategy `enumStrat` on the term `enum(3,6)` is noted

`(enumStrat)enum(3,6)`

and produces four results: `3, 4, 5` and `6`.

2.4 Constructors for strategies

On the simple previous example, 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, the ELAN 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 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.
- The strategy **iterate***(S) is similar to **repeat***(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 class of strategies called elementary strategies. As illustrated in the previous example, 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 $(S)\ t$.

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

```
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
```

One can also generalise variables to patterns, i.e. terms with variables.
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 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 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 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.

2.6 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. The next program shows how to define sets in ELAN, with an Associative Commutative operator `U`, and how to enumerate elements of a set, using the `extract` operation and the 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
[R] extract( (i) U s )  => i end
end
```

Application of the rule `[R]` to `extract(emptySet U (1) U (2) U (3) U (4) U (5))` returns one of `1`, `2`, `3`, `4` and `5`. Application of the strategy `dk(R)` 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 Parameterised and recursive strategies

The class of strategies introduced above does not allow to define strategies with parameters, nor strategies which are recursive. However, these recursive and parameterised strategies may be defined by operators and rewrite rules on an extended set of sorts. Strategies are applied to terms or to other strategies via an explicit application operator, which gives to ELAN a flavor of higher-order language. Details and examples can be found in [Bor98,BCD+99].

3 Object-oriented extension of ELAN

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 object programming, two kinds of languages can be distinguished: class-based languages like Simula [BDMN79], C++, Eiffel [Mey92] or Java [AG96], and object-based languages like Smalltalk [GR83] also studied by [AC96] or [Cas97], where classes are viewed as objects. The concepts integrated in ELAN are selected features of an object-based language: they offer the capability to define classes of objects with attributes and methods. The objects have attributes with read/write values and methods can be revealed or hidden by modifying their access for each object.

According to the object-based approach followed here, a class is an object composed of methods. In order to use objects in a program, the notion of *message passing* is implemented: sending a message to an object consists in attempting to apply a corresponding method on this object. If the method is defined for the object, it is applied on it. As classes are objects, sending a particular message called *new* to the class creates a new object of this class. Sending a message to an object, i.e. calling the corresponding method, is uniformly noted *object.method*.

3.1 Attributes and methods

Let us first define of a class with its methods. Each method has a name and a body which is an instruction (composed of tests and method calls) or a sequence of instructions. Attributes are specific methods whose bodies are values.

For instance, let us consider a class Point that defines objects with only one attribute called X, of sort *int*, initialised to the value 0. This is written in an object module as:

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

An attribute is a method with no argument and whose body is a value. The syntax [1] for the definition of the attributes is as follows:

<*attributes*> **::= attributes** <*list attributes*>

<*list attributes*> **::=** <*attribute name*>**:**<*sort*>**=**<*init value*> |
<*attribute name*>**:**<*sort*>**=**<*init value*><*list attributes*>

Additionally to attributes, different kinds of methods are distinguished:

- methods used to get the current value of an attribute;
- methods used to update the value of an attribute;
- a particular method whose purpose is to create a new object in the class;
- all other methods which do not enter in the previous categories.

The general syntax of a method is:

<*method*> **::= method** <*method name*> [(<*parameters*>)]
for <*sort result*> [<*variables*>] < <*method body*> >

where variables are local variables that are used in the body of the method which is detailed in the following. A method can also have parameters that are defined by a formal name and a sort.

[1] The syntax is given in a BNF style where < *non terminal* > denotes non terminal symbol, **terminal** stands for terminal symbols, [*options*] for optional arguments and *choice*1 | *choice*2 denotes alternative. We give here only a small part of the syntax.

Let us consider the two first kinds of methods. Indeed, when an attribute a is defined with an initial value a_I for an object $Object$, two methods are provided: the first one, called $Geta$, gets the value associated to the attribute a of $Object$; the second one, $Seta(\_)$, with an argument called $Value$, sets to $Value$ the value of the attribute a of $Object$. These methods are in general omitted in the definition of the class, because they can be defined in a canonical way, as explained later on.

In each class, a specific method called *new* is defined which constructs a new object of the class, whose attributes are initialised with default values. This object is defined as having access to all methods defined in the class. The *new* method is also automatically defined when a class is declared.

A method m is always associated to a class and called on an object of this class. This is written as $Object.m$. In a method body, in order to refer to the object to which this method is applied, the **self** keyword is introduced. This is illustrated in the following example, where the method `GetX` associated to the attribute `X` is explicitly defined:

```
class PointWithMethods
 attributes X:int = 0
 method GetX for int <self.X>
End
```

A parameterised method is called with actual values for the parameters. For instance, considering the class $Point$ previously defined, a method named $Reinit$ is introduced in order to set the value of the attribute X to 0. This is written in a new class $PointReinit$ as:

```
class PointReinit
 attributes X:int = 0
 method Reinit for PointReinit <self.SetX(0)>
End
```

The body of this method is only composed by one method call. In general, a body can be more complex.

Tests are also allowed in a method body. The keyword **if** is followed by a boolean expression which has to be evaluated into *true* in order to continue the execution of the method. If the previous $Reinit$ method can only be applied when the value of the attribute X is greater than 100, it is defined as follows:

```
class PointReinit
 attributes X:int = 0
 method Reinit for PointReinit <if (self.GetX)>100 ; self.SetX(0)>
End
```

A method body may involve operators that are not defined as methods in an object module. For instance, let us consider the following example $PointTranslation$ where the method $GetXTranslated$ takes an integer and returns the value of the attribute X added to the given parameter:

```
class PointTranslation
 attributes X:int = 0
 method GetXTranslated(N:int) for int <self.GetX + N>
End
```

Local assignments may also be defined. Let us consider the *PointTranslation* class modified with a local assignment which captures in the variable I previously defined the result of a method call:

```
class PointTranslation
 attributes X:int = 0
 method Translation(N:int) for PointTranslation
  I : int
 <I:= self.GetX ; self.SetX(I + N)>
End
```

A local assignment consists in defining a local variable which can be used in the following instructions composing the method body. These variables can be used as parameters in method calls or in the tests.

Given already defined methods, other methods can be written by composing several methods: the body is then a sequence of calls of methods separated by ";".

Thus, in general, the whole syntax of the body of a method is:

<method body> **::=** *<method call>* |
if *<test>* **;** *<method call>* |
<assignment> **;** *<method body>* |
<method body> **;** *<method body>*

<assignment> **::=** *<variable>* **:=** *<method call>*

<method call> **::=** *<object ident>* **.** *<method name>*[(*<parameters values>*)]

3.2 Inheritance

Each object, with the associated methods, is defined in an object module. Inheritance is provided in OModules in order to construct a hierarchy of class definitions. When a class A inherits a class B (A is a subclass of B), this is written:

```
class A
 from B
End
```

The new class A inherits all attributes and all methods that are defined in the superclass B. New attributes and new methods defined in the subclass are not visible from the superclass.

For instance, let us consider a class *Segment* which inherits the class *Point* by adding a new attribute Y of sort integer initialised to 0:

```
class Segment
 from Point
 attributes Y:int = 0
End
```

4 A rewriting calculus for ELAN with objects

In order to give a formal semantics to this object-oriented extension of ELAN, we study in this section how to restrict the rewriting calculus presented in [CK99a]. The ρ-calculus is briefly presented and we show how classes and objects encoded in the calculus. Methods are also described in this formalism. Some restrictions on the calculus are then introduced by distinguishing mutable attributes and by restricting methods to be defined by rewrite rules which are not changed dynamically. A translation τ is finally defined which makes correspond to any term an associated ρ-term.

4.1 The ρ-calculus

The ρ-calculus [CK99a,CK99b,Cir00] provides a powerful calculus that encompasses in particular both λ-calculus and term rewriting. The syntax of the ρ-calculus is defined by:

$$\begin{array}{rcll} t & ::= & a \mid Y \mid t \to t \mid t \bullet t \mid & \text{plain terms} \\ & & null \mid t, t & \text{structured terms} \end{array}$$

The symbols t, l, r range over the set $\mathcal{T}$ of terms, the symbols $S, Y, Z, \ldots$ range over the set $\mathcal{V}$ of variables, the symbols $a, \ldots, m, 0, 1, \ldots, null, kill, \ldots$ range over the set $\mathcal{C}$ of constants of fixed arity. The symbol $\triangleq$ denotes equality by definition. A rewrite rule $l \to r$ is an abstractor, whose left-hand side determines the bound variables and some contextual information. The application of a ρ-term on another ρ-term is represented using the $\bullet$ operator which is left-associative. The ρ-terms can be grouped together into a structure built using the "," operator. According to the theory behind this operator, different structures can be obtained, as for example a structured list obtained with an associative "," operator, or a set structure with an associative, commutative and idempotent theory. The expression $null$ denotes the empty structure. Priorities are associated to these operators and the $\bullet$ operator has a higher priority than the $\to$ one whose priority is also higher than the "," one.

For a given theory $\mathbb{T}$ describing the structured ρ-terms, the operational semantics is defined by the following computational rules:

$$(\rho)\ (t_1 \to t_2) \bullet t_3 \mapsto_{\mathbb{T}} \begin{cases} null & (n = 0) \\ \sigma_1 t_2, \ldots, \sigma_n t_2 & (1 \leq n \leq \infty) \\ \text{where } \sigma_i \in Sol(t_1 \langle\langle_{\mathbb{T}} t_3) \end{cases}$$

$$(\epsilon)\quad (t_1, t_2) \bullet t_3 \mapsto_{\mathbb{T}} t_1 \bullet t_3, t_2 \bullet t_3$$

$$(\nu)\quad null \bullet t \mapsto_{\mathbb{T}} null.$$

The central idea is that the application of a rewrite rule $t_1 \to t_2$ at the root (also called top) position of a term t_3, consists in computing all possible solutions of the matching equation $(t_1 \ll_{\mathbb{T}} t_3)$ in the theory $\mathbb{T}$ and to apply all the substitutions from the list returned by the function $Sol(t_1 \ll_{\mathbb{T}} t_3)$ to the term t_2. When there is no solution for the matching equation $(t_1 \ll_{\mathbb{T}} t_3)$, the special constant *null* is obtained as result of the application. Notice that there could be an infinity of solutions to the matching problem $(t_1 \ll_{\mathbb{T}} t_3)$ in some theories [FH86].

It is important to remark that if t_1 is a variable, then the (ρ) rule corresponds exactly to the β-rule of the lambda calculus.

The (ϵ) rule dispatches the application inside the structure induced by the "," operator. The (ν) rule takes into account the special case where this structure contains no one element, i.e. when *null* is applied to a term.

When the theory $\mathbb{T}$ for "," is clear from the context, its denotation is omitted. In the following, we choose for $\mathbb{T}$ the AC theory.

4.2 The ρ-calculus for objects

In the ρ-calculus, objects are naturally encoded as structured ρ-terms. A few definitions must be introduced for a better readability. The function $kill_m$ suppresses from a structured ρ-term the rewrite rule whose left-hand side is m:

$$kill_m \ : \ \mathcal{T} \mapsto \mathcal{T} \ \triangleq \ (X, m \to Y) \to X$$

Then some aliases are adopted in this formalism:

$t_1 \cdot t_2$	$\triangleq t_1 \bullet t_2 \bullet t_1$	$self - application$
$t(t_1 \dots t_n)$	$\triangleq t \bullet t_1 \dots \bullet t_n$	$function - application$
$t_1 \cdot m := t_2$	$\triangleq kill_m(t_1), m \to t_2$	*method update*

An object of the class $Point$ defined by:

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

is represented in the ρ-calculus by the ρ-term where the implicit methods $GetX$ and $Setx$ have been clarified in this formalism:

$$\begin{aligned} P \triangleq\ & X \to S \to 0, \\ & GetX \to S \to S \cdot X, \\ & SetX \to S \to N \to S \cdot X \ := \ S' \to N \end{aligned}$$

An object P of class $Point$ consists of the attribute X initialised to 0 and the two methods $GetX$ and $SetX$ which respectively returns the corresponding value and updates the value of X. The abstraction by the bound variable S when defining a method or an attribute means that an implicit parameter of any method represents the **self** object. However, for an attribute, the abstraction on S is never used in the body of this particular method. The bound variable S' is just a new name for the self abstraction variable. Let us illustrate the application of method $SetX$ with the new value 7 on the previous object P:

$$
\begin{aligned}
P \cdot SetX(7) &\triangleq P \cdot SetX \bullet 7 \\
&\triangleq P \bullet SetX \bullet P \bullet 7 \\
&\mapsto (S \to N \to S \cdot X \ := \ S' \to N) \bullet P \bullet 7 \\
&\mapsto (N \to P \cdot X \ := \ S' \to N) \bullet 7 \\
&\mapsto P \cdot X \ := \ S' \to 7 \\
&\triangleq kill_X(P) \ , \ X \to S' \to 7 \\
&\triangleq (GetX \to S \to S \cdot X, \\
&\qquad SetX \to S \to N \to S \cdot X \ := \ S' \to N) \ , \ X \to S' \to 7 \\
&\mapsto X \to S' \to 7, \\
&\qquad GetX \to S \to S \cdot X, \\
&\qquad SetX \to S \to N \to S \cdot X \ := \ S' \to N
\end{aligned}
$$

In the ρ-calculus, as classes are seen as objects, the class $Point$ is defined by a ρ-term:

$$
\begin{aligned}
Point \triangleq\ & new \to S \to \\
& \qquad (X \to S' \to (S \cdot X) \bullet S', \\
& \qquad GetX \to S' \to (S \cdot GetX) \bullet S', \\
& \qquad SetX \to S' \to (S \cdot SetX) \bullet S'), \\
& X \to S \to S' \to 0, \\
& GetX \to S \to S' \to S' \cdot X, \\
& SetX \to S \to S' \to N \to S' \cdot X \ := \ S'' \to N
\end{aligned}
$$

Objects and classes are structured terms of the ρ-calculus. A class is defined as:

$$
\begin{aligned}
class \ \triangleq \ & new \to S \to object, \\
& meth_1 \to S \to S' \to \ldots \to body_1, \\
& \ldots \\
& meth_m \to S \to S' \to \ldots \to body_m
\end{aligned}
$$

An object of this class is defined as:

$$
\begin{aligned}
object \ \triangleq \ & meth_1 \to S \to \ldots \to body_1, \\
& \ldots \\
& meth_m \to S \to \ldots \to body_m
\end{aligned}
$$

A specific method of each class is the method new which defines an object of this class. In order to illustrate the behaviour of the calculus, the creation of a new object of object representing the class Point, obtained by calling the method new on the class $Point$, is detailed thereafter. The symbol $\equiv_\alpha$ denotes the variable renaming:

$$
\begin{aligned}
& Point \cdot new \ \triangleq \ Point \bullet new \bullet Point \\
\triangleq \ & (S \to (X \to S' \to (S \cdot X) \bullet S', \\
& \qquad GetX \to S' \to (S \cdot GetX) \bullet S', \\
& \qquad SetX \to S' \to (S \cdot SetX) \bullet S')) \bullet Point
\end{aligned}
$$

$$
\begin{array}{ll}
\mapsto & X \to S' \to (Point \cdot X) \bullet S', \\
& GetX \to S' \to (Point \cdot GetX) \bullet S', \\
& SetX \to S' \to (Point \cdot SetX) \bullet S' \\
\triangleq & X \to S' \to ((Point \bullet X) \bullet Point) \bullet S', \\
& GetX \to S' \to ((Point \bullet GetX) \bullet Point) \bullet S', \\
& SetX \to S' \to ((Point \bullet SetX) \bullet Point) \bullet S' \\
\mapsto & X \to S' \to ((S \to S' \to 0) \bullet Point) \bullet S', \\
& GetX \to S' \to ((S \to S' \to S' \cdot X) \bullet Point) \bullet S', \\
& SetX \to S' \to ((S \to S' \to N \to S' \cdot X \ := \ S'' \to N) \bullet Point) \bullet S' \\
\mapsto & X \to S' \to (S' \to 0) \bullet S', \\
& GetX \to S' \to (S' \to S' \cdot X) \bullet S', \\
& SetX \to S' \to (S' \to N \to S' \cdot X \ := \ S'' \to N) \bullet S' \\
\mapsto & X \to S' \to 0, \\
& GetX \to S' \to S' \cdot X, \\
& SetX \to S' \to N \to S' \cdot X \ := \ S'' \to N \\
\equiv_\alpha & X \to S \to 0, \\
& GetX \to S \to S \cdot X, \\
& SetX \to S \to N \to S \cdot X \ := \ S' \to N
\end{array}
$$

A first approach to implement the object-oriented extension of ELAN, is to implement the ρ-calculus for objects previously described. This approach has been followed in [Cir00]. Rules are then considered as ρ-terms and can be changed during evaluation by rewriting. Methods can be dynamically changed too. This gives rise to a very expressive language, but the implementation needs to encode in particular how to apply a rule (as ρ-term) to another ρ-term and does not take advantage of the fact that rewrite rules can be executed very efficiently by the ELAN rewrite engine.

So we choose here another approach where the application of a method is defined with ELAN rewrite rules and executed by the ELAN rewrite engine. Since ELAN rewrite rules cannot be dynamically updated at evaluation time, this imposes some restrictions on the calculus. However we will see later that these restrictions are quite reasonable in the context of our applications where modifications of the attribute values are the main operations that are performed.

4.3 An algebraic encoding of objects

For the reasons explained before, we choose to distinguish 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 advantage is that 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. An object has the form:

$$[a_1(v_1), ..., a_n(v_n), meth_1, ..., meth_m]$$

and corresponds to the ρ-term:

$$\begin{aligned} object \triangleq\ & a_1 \to S \to v_1, \\ & \ldots \\ & a_n \to S \to v_n, \\ & meth_1 \to S \to \ldots \to body_1, \\ & \ldots \\ & meth_m \to S \to \ldots \to body_m \end{aligned}$$

where n is the number of attributes a_i of sort t_i with their corresponding v_i values and m the number of methods $meth_j$. The bodies of methods are defined by rewrite rules in the module defining the class. Let us explain how to build the associated rewrite rules. Let us consider a method m with the self parameter S, the formal parameters p_1 to p_n, and a body b. The left-hand side is always $m(S, p_1, \ldots, p_n)$. The right-hand side depends on the form of the body. Let us consider an operator $build - rhs(\_, \_)$ which takes a body and a reference to an object. Initially, the right-hand side is computed by $build - rhs(b, S)$.

- if b is a method call of the form $o.m'(p'_1, \ldots, p'_n)$, then:

$$build - rhs(o.m'(p'_1, \ldots, p'_n), O) \ = \ m'(O, p'_1, \ldots, p'_n)$$

- if b is a method, involving a given operator f, of the form $f(p'_1, \ldots, p'_n)$, then:

$$build - rhs(f(p'_1, \ldots, p'_n), O) \ = \ f(p'_1, \ldots, p'_n)$$

- if b is a test followed by a method call of the form $if\ t\ ;\ o.m'(p'_1, \ldots, p'_n)$, then:

$$build - rhs(if\ t\ ;\ o.m'(p'_1, \ldots, p'_n), O) \ = \ m'(O, p'_1, \ldots, p'_n) \ \textbf{if} \ \ t$$

- if b is an assignment followed by a method body of the form:

$$variable \ := \ f(p'_1, \ldots, p'_n)\ ;\ body$$

then:

$$build - rhs(variable \ := \ f(p'_1, \ldots, p'_n)\ ;\ body, O) \ =$$

$$B' \ \textbf{where} \ \ variable := ()f(p'_1, \ldots, p'_n) \ \textbf{where} \ \ B' := ()build - rhs(body, O)$$

- if b is the composition of two method bodies $b1\ ;\ b2$, then:

$$build - rhs(b1\ ;\ b2, O) \ =$$

$$O_2 \textbf{where} \ \ O_1 := ()build - rhs(b1, O) \ \textbf{where} \ \ O_2 := ()build - rhs(b2, O_1)$$

Objects are defined by a structure mixing mutable attributes and references to methods defined by rewrite rules. A class is also defined as an object and has the form:

$$[a_1(vi_1), ..., a_n(vi_n), meth_1, ..., meth_m, new]$$

It corresponds to the ρ-term:

$$\begin{aligned}
class \triangleq\ & a_1 \to S \to vi_1, \\
& \ldots \\
& a_n \to S \to vi_n, \\
& meth_1 \to S \to \ldots \to body_1, \\
& \ldots \\
& meth_m \to S \to \ldots \to body_m, \\
& new \to S \to (a_1 \to S' \to (S \cdot a_1) \bullet S', \\
& \qquad \ldots \\
& \qquad a_n \to S' \to (S \cdot a_n) \bullet S', \\
& \qquad meth_1 \to S' \to (S \cdot meth_1) \bullet S', \\
& \qquad \ldots \\
& \qquad meth_m \to S' \to (S \cdot meth_m) \bullet S')
\end{aligned}$$

where each attribute a_i is initialised with the corresponding initial value vi_i. The reader can refer to the previous example of the class *Point*, which details the application of a method *new* on an object of such a class in order to create a new object of this class.

In order to define an algebraic encoding of objects, let us define the rewrite theory $\mathcal{R}$ composed of a signature Σ and a set of rules R. The signature, given in an ELAN syntax, is the following one:

```
operators global
  [@]    : (Methods) Object;
  @,@    : (Methods Methods) Methods     (AC);
  @      : (method) Methods;

  @(@)   : (MName MBody) method;
  @      : (MName) method;
  m(@)   : (MBody) method;

  m      : MName;
  Getm   : MName;
  Setm   : MName;

  add(@,@)     : (Object method) Object;
  access(@,@)  : (Object MName) MBody;
  kill(@,@)    : (Object MName) Object;
  Getm(@)      : (Object) MBody;
  Setm(@,@)    : (Object MBody) Object;
  new(@)       : (Object) Object;
end
```

The first operator constructs from a term of sort `Methods` a term of sort `Object`. A term of sort `Methods` is a multiset of terms of sort `method` with the constructor "," which is declared AC. A method is defined either as a name followed by its body (for an attribute `m` with its value) or simply by the name of the method. Then, an operator `add` takes a term of sort `Object` and another of sort `method` and returns a term of sort `Object`. Other useful operations are the access to the value (resp. the body) of an attribute (resp. a method) and the deletion of a method from the list of methods. To access to the value of an attribute, the operator `access` is introduced, while `kill` removes a method from the list of methods. Generic operators `Getm` and `Setm` are defined to describe the primitive operations on an object: getting the value of an attribute `m` by `Getm` and changing its value by `Setm`.

The following set of rules R defines each operator of this signature. The rule for `add` defines how the initial object `[M]` is extended to `[m,M]` with a new method `m`:

```
rules for Object
 LM : Methods;
 me : method;
global
[] add([LM],me)  => [me,LM] end
end
```

The `access` operator takes a term of sort `Object` and another one of sort `MName` and returns the body corresponding to the method whose name is given as second argument.

```
rules for MBody
 M  : MName;
 B  : MBody;
 LM : Methods;
global
[] access([M(B),LM],M) => B end
end
```

The `kill` operator takes a term of sort `Object` and another one of sort `MName` and returns the object minus the corresponding method. The rules defining this operator are therefore:

```
rules for Object
 M  : MName;
 LM : Methods;
 B  : MBody;
global
[] kill([M(B),LM],M) => [LM] end
[] kill([M,LM],M)    => [LM] end
end
```

Given `add`, `access` and `kill`, for each object with an attribute `m`, rules for `Getm` and `Setm` are defined by:

```
rules for MBody
 o : Object;
global
[] Getm(o)   => access(o,m) end
end

rules for Object
 o : Object;
 B : MBody;
global
[] Setm(o,B) => add(kill(o,m),m(B)) end
end
```

To each attribute a_i corresponds a constant `a_i` in the signature. Initial values vi_j for attributes are constants `vi_j` of sort `MBody`. To each method m_i corresponds a constant `m_i` and an operator `m_i(...)` in the signature. Then, for objects representing classes, an additional operator `new` is defined:

```
rules for Object
  o  : Object;
global
[] new(o) => [a_1(vi_1),...,a_n(vi_n),m_1,....,m_n] end
end
```

A new object of the class is thus created with initial values for each attribute and with all methods of the class.

For instance, let us consider an object of the class *Point* with the attribute X of sort *int* set to 5 and the two methods $GetX$ and $SetX$. This object is implemented as:

```
[X(5) , SetX , GetX]
```

with respective rules for defining the methods `SetX` and `GetX`:

```
rules for int
 o : Point;
global
[] GetX(o)   => access(o,X) end
end

rules for Point
 o : Point;
 V : MBody;
global
[] SetX(o,V) => add(kill(o,X),X(V)) end
end
```

In the rewrite theory $\mathcal{R}$, an object is represented as a term of sort `Object`. It is not difficult to prove that the set of rules in R is terminating and confluent. So each term t of sort `Object` has a normal form denoted by $NF(t)$. It is also easy to check that this normal form is a multiset which contains, for each attribute

m a 3-tuple $[m(b), Getm, Setm]$, and for each method which is not an attribute only the method name. A generic form of a term t_0 of sort `Object` in normal form is thus written in the following:

$$t_0 = [m(b)\ ,\ Getm\ ,\ Setm\ ,\ LM]$$

where LM is a multiset of terms of sort `method` of one of the following form: $m', m'(b'), Getm', Setm'$.

Let us consider the translation τ which maps such a term

$$t_0 = [m(b)\ ,\ Getm\ ,\ Setm\ ,\ LM]$$

in the rewrite theory $\mathcal{R}$ to the ρ-term t_0':

$$\begin{aligned}\tau(t_0) = t_0' = (&m \to S \to b,\\ &Getm \to S \to S \cdot m,\\ &Setm \to S \to N \to S \cdot m := S' \to N,\\ &L)\end{aligned}$$

where L is a multiset of ρ-terms of the form $m' \to S \to body$.

Let us detail τ by giving the associated ρ-terms for a method called m', with the self parameter S and n parameters denoted p_i:

- if the rule defining m' is of the form:

$$[]\ \ m'(S, p_1, \ldots, p_n)\ \ \to\ \ f(S, p_1, \ldots, p_n)$$

  then, the associated ρ-term is:

$$m' \to S \to p_1 \to \ldots \to p_n \to f(S, p_1, \ldots, p_n)$$

- if the rule defining m' is of the form:

$$[]\ \ m'(S, p_1, \ldots, p_n)\ \ \to\ \ f(S, p_1, \ldots, p_n, o)\ \ \textbf{where}\ \ o := ()b$$

  then, the associated ρ-term is:

$$m' \to S \to p_1 \to \ldots \to p_n \to (o \to f(S, p_1, \ldots, p_n, o)) \bullet b$$

- if the rule defining m is of the form:

$$[]\ \ m'(S, p_1, \ldots, p_n)\ \ \to\ \ b\ \textbf{if}\ \ c$$

  then, the associated ρ-term is:

$$m' \to S \to p_1 \to \ldots \to p_n \to (True \to b) \bullet c$$

- if two rules are associated to the same operator m':

$$\begin{aligned}[]\ \ m'(S, p_1, \ldots, p_n)\ \ &\to\ \ b_1\\ []\ \ m'(S, p_1', \ldots, p_n')\ \ &\to\ \ b_2\end{aligned}$$

  then, the associated ρ-term is:

$$\begin{aligned}m' \to S \to P_1 \to \ldots \to P_n \to (\ &m'(S, p_1, \ldots, p_n) \to b_1,\\ &m'(S, p_1', \ldots, p_n') \to b_2) \bullet m'(S, P_1, \ldots, P_n).\end{aligned}$$

The following proposition relates evaluation by rewriting in the rewrite theory $\mathcal{R}$ and computation in the ρ-calculus:

Proposition 1. *Let $\equiv$ be the transitive, reflexive and symmetric closure of the relation $\mapsto$ in the ρ-calculus. For any terms t, t_0 of sort* `Object` *in $\mathcal{R}$:*

- $\tau(NF(add(t_0, m_1))) \equiv \tau(NF(t_0)), \tau(m_1)$
- $\tau(NF(access(t_0, m))) \equiv \tau(NF(t_0)) \cdot \tau(m)$
- $\tau(NF(kill(t_0, m))) \equiv kill_m(\tau(NF(t_0)))$
- $\tau(NF(Getm(t_0))) \equiv \tau(NF(t_0)) \cdot Getm$
- $\tau(NF(Setm(t_0, V))) \equiv \tau(NF(t_0)) \cdot Setm(V)$
- $\tau(NF(new(t))) \equiv \tau(NF(t)) \cdot new$

Proof. Let $NF(t_0) = [m(b)\ ,\ Getm\ ,\ Setm\ ,\ LM]$ and

$$\begin{aligned}\tau(NF(t_0)) = t_0' = (&m \to S \to b,\\ &Getm \to S \to S \cdot m,\\ &Setm \to S \to N \to S \cdot m := S' \to N,\\ &L)\end{aligned}$$

where $L = \tau(LM)$. Let us define $\to_R$ as the rewriting relation on terms defined by application of a rule in R and $\overset{*}{\to}_R$ as the reflexive and transitive closure of $\to_R$.

- For any method m_1,

  $$\begin{aligned}add(t_0, m_1) &\overset{*}{\to}_R add([m(b)\ ,\ Getm\ ,\ Setm\ ,\ LM], m_1)\\ &\to_R [m(b)\ ,\ Getm\ ,\ Setm\ ,\ LM\ ,\ m_1] = t_1.\end{aligned}$$

  On the other hand, in the ρ-calculus:

  $$\begin{aligned}\tau(NF(t_0)), \tau(m_1) &\triangleq (m \to S \to b,\\ &\quad Getm \to S \to S \cdot m,\\ &\quad Setm \to S \to N \to S \cdot m := S' \to N,\\ &\quad \tau(LM)),\\ &\quad \tau(m_1)\\ &\mapsto (m \to S \to b,\\ &\quad Getm \to S \to S \cdot m,\\ &\quad Setm \to S \to N \to S \cdot m := S' \to N,\\ &\quad \tau(LM), \tau(m_1))\\ &\triangleq \tau(t_1)\end{aligned}$$

  So $\tau(NF(add(t_0, m_1))) \equiv \tau(NF(t_0)), \tau(m_1)$.
- For any method m,

  $$\begin{aligned}access(t_0, m) &\overset{*}{\to}_R access([m(b)\ ,\ Getm\ ,\ Setm\ ,\ LM], m)\\ &\to_R b\end{aligned}$$

On the ρ-term $t'_0 \cdot m = \tau(t_0) \cdot m$, the following evaluation can be performed in the ρ-calculus:

$$\begin{aligned}
t'_0 \cdot m &\triangleq t'_0 \bullet m \bullet t'_0 \\
&\mapsto (m \to S \to b, \\
&\quad Getm \to S \to S \cdot m, \\
&\quad Setm \to S \to N \to S \cdot m := S' \to N, \\
&\quad LM) \bullet m \bullet t'_0 \\
&\mapsto (S \to b) \bullet t'_0 \\
&\mapsto b
\end{aligned}$$

So $\tau(NF(access(t_0, m))) \equiv \tau(NF(t_0)) \cdot \tau(m)$.

– For any method m,

$$kill(t_0, m) \xrightarrow{*}_R kill(NF(t_0), m) \to_R [Getm\ ,\ Setm\ ,\ LM]$$

On the other hand, in the ρ-calculus:

$$\begin{aligned}
kill_m(\tau(NF(t_0)) \triangleq\ & (Getm \to S \to S \cdot m, \\
& Setm \to S \to N \to S \cdot m := S' \to N, \\
& L)
\end{aligned}$$

So $\tau(NF(kill(t_0, m))) \equiv kill_m(\tau(NF(t_0))$.

– On one hand

$$\begin{aligned}
Getm(t_0) &\xrightarrow{*}_R Getm(NF(t_0)) \\
&\to_R Getm([m(b)\ ,\ Getm\ ,\ Setm\ ,\ LM]) \\
&\to_R access([m(b)\ ,\ Getm\ ,\ Setm\ ,\ LM], m) \\
&\to_R b
\end{aligned}$$

On the other hand, in the ρ-calculus:

$$\begin{aligned}
t'_0 \cdot Getm &\triangleq t'_0 \bullet Getm \bullet t'_0 \\
&\triangleq (m \to S \to b, \\
&\quad Getm \to S \to S \cdot m, \\
&\quad Setm \to S \to N \to S \cdot m := S' \to N, \\
&\quad L) \bullet Getm \bullet t'_0 \\
&\mapsto (S \to S \cdot m) \bullet t'_0 \\
&\mapsto t'_0 \cdot m \\
&\triangleq t'_0 \bullet m \bullet t'_0 \\
&\triangleq (m \to S \to b, \\
&\quad Getm \to S \to S \cdot m, \\
&\quad Setm \to S \to N \to S \cdot m := S' \to N, \\
&\quad L) \bullet m \bullet t'_0 \\
&\mapsto (S \to b) \bullet t'_0 \\
&\mapsto b
\end{aligned}$$

So $\tau(NF(Getm(t_0))) \equiv \tau(NF(t_0)) \cdot Getm$.

– On one hand:

$$\begin{aligned}
Setm(t_0, V) &\xrightarrow{*}_R Setm(NF(t_0), V) \\
&\to_R Setm([m(b)\ ,\ Getm\ ,\ Setm\ ,\ LM], V) \\
&\to_R add(kill([m(b)\ ,\ Getm\ ,\ Setm\ ,\ LM], m), m(V)) \\
&\to_R add([Getm\ ,\ Setm\ ,\ LM], m(V)) \\
&\to_R [m(V)\ ,\ Getm\ ,\ Setm\ ,\ LM]
\end{aligned}$$

On the other hand, in the ρ-calculus:

$$\begin{aligned}
t'_0 \cdot Setm(V) &\triangleq t'_0 \bullet Setm \bullet t'_0 \bullet V \\
&\triangleq (m \to S \to b, \\
&\quad Getm \to S \to S \cdot m, \\
&\quad Setm \to S \to N \to S \cdot m := S' \to N, \\
&\quad L) \bullet Setm \bullet t'_0 \bullet V \\
&\mapsto (S \to N \to S \cdot m := S' \to N) \bullet t'_0 \bullet V \\
&\mapsto (N \to t'_0 \cdot m := S' \to N) \bullet V \\
&\mapsto t'_0 \cdot m := S' \to V \\
&\triangleq kill_m(t'_0), m \to S' \to V \\
&\mapsto (Getm \to S \to S \cdot m, \\
&\quad Setm \to S \to N \to S \cdot m := S' \to N, \\
&\quad L), m \to S' \to V \\
&\mapsto (m \to S' \to V, \\
&\quad Getm \to S \to S \cdot m, \\
&\quad Setm \to S \to N \to S \cdot m := S' \to N, \\
&\quad L)
\end{aligned}$$

So $\tau(NF(Setm(t_0, V))) \equiv \tau(NF(t_0)) \cdot Setm(V)$.

– On one hand:

$new(t) \xrightarrow{*}_R [m(vi_m)\ ,\ Getm\ ,\ Setm\ ,\ LM]$

On the other hand, in the ρ-calculus:

$$\begin{aligned}
t' \cdot new &\triangleq (m \to S \to S' \to vi_m, \\
&\quad Getm \to S \to S' \to S \cdot m, \\
&\quad Setm \to S \to S' \to N \to S \cdot m := S' \to N, \\
&\quad L, \\
&\quad new \to S \to \\
&\qquad (m \to S' \to (S \cdot X) \bullet S', \\
&\qquad Getm \to S' \to (S \cdot GetX) \bullet S', \\
&\qquad Setm \to S' \to (S \cdot SetX) \bullet S', \\
&\qquad L1)) \cdot new \\
&\triangleq (m \to S \to S' \to vi_m, \\
&\quad Getm \to S \to S' \to S \cdot m, \\
&\quad Setm \to S \to S' \to N \to S \cdot m := S' \to N, \\
&\quad L, \\
&\quad new \to S \to \\
&\qquad (m \to S' \to (S \cdot X) \bullet S', \\
&\qquad Getm \to S' \to (S \cdot GetX) \bullet S', \\
&\qquad Setm \to S' \to (S \cdot SetX) \bullet S', \\
&\qquad L1)) \bullet new \bullet t' \\
&\mapsto^* m \to S' \to vi_m, \\
&\quad Getm \to S \to S \cdot m, \\
&\quad Setm \to S \to N \to S \cdot m := S' \to N, \\
&\quad LM
\end{aligned}$$

The last reduction is not further detailed as it follows the same principle than the one presented in the example for the class $Point$.

So $\tau(NF(new(t)) \equiv \tau(NF(t)) \cdot new$ □

As shown in the proof, the initial term $Getm(t_0)$ is rewritten into b with R. Thus, the initial rule defining `Getm` can be simplified into the equivalent one:

```
rules for MBody
  LM : Methods;
  B  : MBody;
global
[] Getm([m(B),LM])   => B end
end
```

Again, as shown in the proof, the initial term $Setm(t_0, V)$ is rewritten into $[m(V)\ ,\ Getm\ ,\ Setm\ ,\ LM]$ with R. Thus, the initial rule defining `Setm` can be simplified into the equivalent one:

```
rules for Object
 LM : Methods;
 B  : MBody;
 V  : MBody;
global
[] Setm([m(B),LM],V)   => [m(V),LM] end
end
```

Let us extend now the definition of the translation τ by setting, for any t_0 of sort `Object` and m, m_1 of sort `method`:

- $\tau(add(t_0, m_1) = \tau(t_0), \tau(m_1)$
- $\tau(access(t_0, m)) = \tau(t_0) \cdot \tau(m)$
- $\tau(kill(t_0, m)) = kill_m(\tau(t_0))$
- $\tau(Getm(t_0)) = \tau(t_0) \cdot Getm$
- $\tau(Setm(t_0, V)) = \tau(t_0) \cdot Setm(V)$
- $\tau(new(t_0)) = \tau(t_0) \cdot new$

This leads to the following result, that relates normalisation in the rewrite theory $\mathcal{R}$ and evaluation in the ρ-calculus:

Theorem 2. *For every term t in $\mathcal{R}$, $\tau(NF(t)) \equiv \tau(t)$.*

Proof. The proof is by induction on the structure of the term t and Proposition 1. Let us detail the proof for a term t written $add(t_0, m_1)$. We have:

$$\begin{aligned} \tau(NF(add(t_0, m_1))) &= \tau(NF(t_0)), \tau(m_1) && \text{by Proposition 2} \\ &= \tau(t_0), \tau(m_1) && \text{by induction} \\ &= \tau(add(t_0, m_1)) && \text{by definition of } \tau \end{aligned}$$

□

Thus, we have defined objects whose attributes have read/write values and whose method accesses can be changed during evaluation thanks to `kill` and `add`. During an execution step, methods can be revealed or hidden by modifying the accesses for each object.

5 Compilation in ELAN

In order now to execute programs written in this object extension of ELAN, we adopt the following method: object modules are translated automatically into ELAN modules that can be executed by the ELAN interpreter or compiler. In other words, object-oriented programs are compiled into ELAN code.

For instance, let us consider the program given for the class *Point*:

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

This program is compiled into the *PointClass* ELAN module given in Fig. 1.

```
module PointClass

import global DefClass[Point] SortAttDecl[int]
              GetAtt[Point,X,GetX,int]
              SetAtt[Point,X,SetX,int]
              ClassInit[Point,PointClass];
end

rules for Point
  o   : Point;
global
[] PointClass => [X(0) , GetX , SetX , new] end
[] o.new      => [X(0) , GetX , SetX] end
end
end
```

Fig. 1. ELAN module for the class *Point*

The compiled program takes advantage of the modularity and parameterisation features of ELAN. The ELAN module *PointClass* is composed of importations of parameterised modules and rule declarations. Indeed, several generic modules have been defined in ELAN which just have to be parameterised with the appropriate data in order to generate the good sorts, operators and rules. In this example, some of these modules are introduced:

- The module `DefClass` defines or imports the basic operator definitions for constructing an object of sort `Point`. This sort is given as parameter.
- The module `SortAttDecl` is a generic module which makes the appropriate declarations in order to define that an attribute can take its values in this sort.
- The generic module `GetAtt` (resp. `SetAtt`) declares operators and rules used for getting (resp. setting) the value of an attribute. The parameters are the

name of the class, the name of the attribute, the name of the method and the sort of the attribute values. Here is the generic module `GetAtt` with parameters `Class`, `Att`, `GetAtt` and `SortAtt`:

```
module GetAtt[Class,Att,GetAtt,SortAtt]

import global DefClass[Class] AttributeDecl[Att]
              SortAttDecl[SortAtt]; end

operators global
  GetAtt          : MName;
  @.Att           : (Class) SortAtt;
  GetAtt(@)       : (Class) SortAtt;
end

rules for SortAtt
  LM  : Methods;
  V   : SortAtt;
global
[] [GetAtt,LM].Att          => GetAtt([GetAtt,LM]) end
[] GetAtt([Att(V),LM])      => V end
end
end
```

This module produces the corresponding importations and operator or rule declarations that are instantiated with the values given for each parameter.

– The last generic module is `ClassInit` declares the operators needed to initialise an object of a class by the method `new`.

Among the two rules of the module `PointClass`, the first one creates the class *Point* as a new object `[X(0) , GetX , SetX , new]` and the second one generates a new object of this class `[X(0) , GetX , SetX]`.

The originality of this compilation process is that it is written itself as an ELAN program. In a more general way, this shows how to use ELAN to produce from a program given in the syntax of a language we want to prototype, an ELAN program that can be interpreted and compiled.

The transformation is defined by five steps whose fourth first ones are entirely written in ELAN. 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. 2.

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

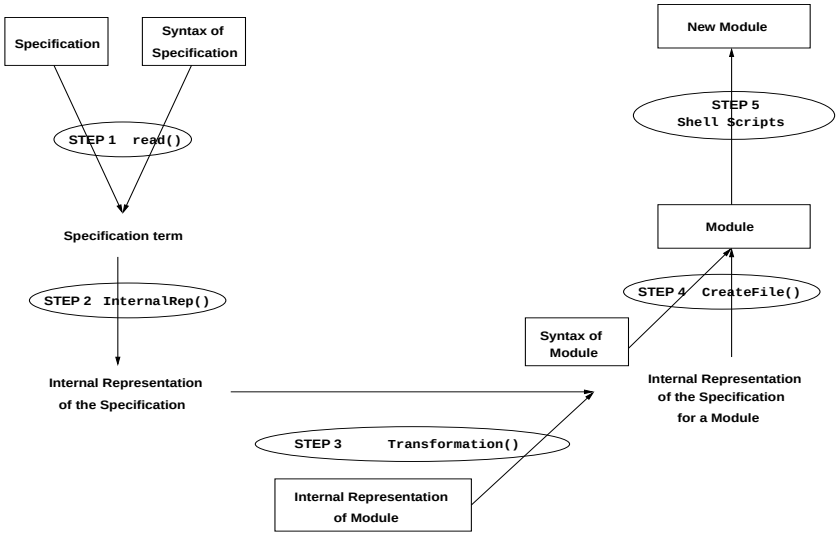

Fig. 2. From a specification to ELAN module

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.

This general transformation scheme is here adapted to the transformation of object module into ELAN modules. More generally, this scheme can be adapted and has been tested for other program transformations and language translations [DK00a].

For performing this compilation 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 compilation process and is expressed as follows:

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

The rule introduces intermediate results `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. Each step is illustrated with the class *Point* example.

Reading the input program

In order to read and parse the specification given in the input program, an operator `GetTermFromSpecif()` is defined in ELAN. It uses an ELAN module in which the syntax of the program given as input is defined. A part of this ELAN file where the syntax of an object module is expressed is presented below:

```
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 @       : (AttributeDeclareList) Attributes;
 @            : (AttributeDeclare) AttributeDeclareList;
 @ @          : (AttributeDeclare AttributeDeclareList) AttributeDeclareList;
 @ ':' @ = @  : (AttributeName SortName InitValue) AttributeDeclare;
...
end
```

The result `Term1` of this first transformation of the input module class `Point` is:

```
class Point imports int; attr X : int = 0 End
```

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, which corresponds to the second step of the transformation. The `InternalRepOfSpecif()` operator produces from `Term1`, the second result `Term2`. This internal representation takes advantage from data structures predefined in ELAN, namely lists, pairs and n-tuples. Here, `Term2` is:

```
[Point,none,int.nil,[X,int,0].nil].nil
```

Transformation of the internal representation

The step corresponding to the `FromInternalRepToNewModules()` operator transforms the internal representation of an object module into the corresponding internal representation of an ELAN module. The result obtained in `Term3` is already very close to the ELAN syntax:

```
module Point
import global SortAttDecl[int] AttributeGet[Point,X,Get X,int]
              AttributeModif[Point,X,Set X,int] int none
              DEFClass[Point] ClassInit[Point,Point Class];End

Rules for Point
LM:Methods;o:Point;
global
[]Point Class=>[X(0),Get X,Set X,new]End
[]new(o)=>[X(0),Get X,Set X]End
End
End.nil
```

Generation of a new module

The last step of the compilation is realised with the `CreateModule()` operator which takes the internal representation of the ELAN module as argument and produces the corresponding file for the set of modules. From the previous `Term3`, which is a list of terms representing modules, several modules are generated, one for each class. In the case of our example, the generated module contains the first (and unique) element of the previous list of modules.

Once these files are built, the last step uses a shell script with `sed` commands to produce a more readable output program and to transform a few reserved keywords that ELAN cannot produce itself. The new module corresponding to the *Point* example is the module given in Fig. 1.

Focus on the transformation of the internal representation

To better understand how this compilation is performed, let us look at the rule which is at the top level of the transformation from the internal representation of the object module into the internal representation of an ELAN module, which is the more complex step of the compilation.

This is a rule that builds a term (M) 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 name of the module is directly obtained as the first item of the structure T. The definition of importations is done by the `GetELANImports` operator dedicated to global importations of modules. New operators and rules are obtained respectively by the `GetELANOperators` and the `GetELANRules` operators. The result term `M` of the rule is built from these subterms `MN`, `MI`, `MO` and `MR`.

```
[] NewClassModules(T) => M.nil
    where MN := () 1-th(T)
    where MI := () import global GetELANImports(T); End
    where MO := () GetELANOperators(T)
    where MR := () GetELANRules(T)
    where M := () module MN MI MO MR End end
```

6 Rules and strategies on objects

As already done with constraints in [DK00b], rules and strategies to express control are now extended to objects. This provides the adequate framework to describe a 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. To take into account this concept in the ρ-calculus, as well as in the algebraic rewrite theory, a new binary symbol $\{\_,\_\}$ is introduced to construct multisets of objects, and is declared as an associative and commutative symbol.

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

6.1 Rules on objects

Programs on objects are sets of rewrite rules on a multiset of objects. Changes in the data base of objects are described through conditional rewrite rules of the form:

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

where $O_1, \dots, O_k, O'_1, \dots, O'_m$ are 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. An object O_i in the left-hand side of the rule has one of the following forms:

$O_i : ClassName_i :: [Att_1(Value_1)\ ,\ \dots\ ,\ Att_n(Value_n)]$
$O_i : ClassName_i$

The first form corresponds to look in the working memory for an object of class $ClassName_i$ whose attributes $Att_1, \dots, Att_n$ have the corresponding values $Value_1, \dots, Value_n$. The order of attributes is irrelevant and some attributes of the objects can be omitted. The second form corresponds to look for any object of class $ClassName$ in the working memory. Once an object is selected, it is associated to a name O_i which denotes this object during the rule application.

Application of this rule succeeds only if the tests **if** t succeed (i.e. is evaluated into $true$) 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.

Methods that are defined in classes can be applied to any object of the data base. In practice, methods are mainly used on modified and new objects because applying methods on context and deleted objects is useless. A few standard notations in object oriented languages are now introduced to be used in the right-hand side, the tests and the local assignments of these rules.

- In order to get the value associated to an attribute `X` of an object `O`, instead of writing it `GetX(O)`, it is written `O.X`.
- In order to modify the value of an attribute `X` of an object `O` into the new value `V`, instead of `SetX(O,V)`, it is written `O(X<-V)`. When several modifications are performed on the same object, instead of calling `SetX1(O,V1)`, then `SetX2(O,V2)` and so on, it is simplified into `O(X1<-V1,...,Xn<-Vn)`.
- In order to apply any method `m` on an object `O`, the standard notation `O.m` is used for any method call. The parameters are given in the same way.

6.2 Translation of rules on objects in ELAN

The rules defined on objects have to be slightly transformed to get ELAN rules. In particular, unspecified arguments of associative and commutative operators have to be captured 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 associative and commutative 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

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

is automatically transformed into the schema of ELAN rule given in Fig. 3.

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

$Vars$
O_j : $ClassName_j$;
X'_i : $ClassName_i$;
$At_1, ..., At_k$: $AttributeList$;
1[Lab] $\{[a_1(v_1)\ ,\ \ldots\ ,\ a_{n_i}(v_{n_i})\ ,\ At_i]\}_{i=[1,\ldots,k]}\ \Rightarrow\ X'_c\ X'_m\ X'_n$
2 **where** $O_j := ()[a_j(v_j)\ ,\ \ldots\ ,\ a_{n_j}(v_{n_j})\ ,\ At_j]$
% *for the k objects of the original rule*
3 [**if** t | **where** l]*
4 **where** $X'_c := ()\ O_{c\in[1,\ldots,k]}$
% *for each context object*
5 **where** $X'_m := ()\ O_{j\in[1,\ldots,k]}[\{a_l \leftarrow v_l\}_{l\in[1,\ldots,N_j]}]$
% *for each modified object*
6 **where** $X'_n := ()\ ClassName_n.new$
% *for each created object*

Fig. 3. Translation of an object rule.

- 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).

6.3 An example

In order to illustrate now the extended language, let us consider a program designing the control of elevators in a building 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 currently processed floor, and the second one is the list of waiting floors that have to be served to unload 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.

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`); the list of floors where it has to stop (the attribute `LStop`).

The sort describing the state of an elevator is called `LiftState`. Terms of this sort are 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 values in $\{0, 1\}$ and if `I`$= 0$, then, the elevator 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 upwards or downwards 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 for applying 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)]
      =>
    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::[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)
```

6.4 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 looks if the direction of the elevator has to be changed and, otherwise, it continues to go upwards or downwards.

```
[] 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` updates all flags `F` from the value 1 to 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 the `MAIN` strategy 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.

7 Conclusion

Related work on object oriented programming based on rewriting logic is presented in [CDE+00,DMT00]. The OMaude system is based on object-oriented modules specifying distributed object systems in which the top structure of the distributed state is an associative and commutative multiset of objects and messages.

Mixing objects, rules and strategies in a same language gives rise to an increased expressive power. The prototype of language presented in this paper confirms this idea. But when writing programs, two main questions are 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. Those two options are currently explored.

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 [Rou88] designed to

prove the consistency of knowledge-based systems. PREPARE [ZN94] is also a system able to detect potential errors of rule-based systems. In [SS95], Schmolze and Snyder have also defined a tool based on the Knuth-Bendix Completion [KB70] 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 [Frü00]. 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.

Acknowledgments: We sincerely thank Horatiu Cirstea, Claude Kirchner and Luigi Liquori for fruitful discussions on the ρ-calculus with objects. Exchanges with the Protheo group at LORIA were also very helpful during this work.

References

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

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

[BCD+99] Peter Borovanský, Horatiu Cirstea, Hubert Dubois, Claude Kirchner, Hélène Kirchner, Pierre-Etienne Moreau, Christophe Ringeissen, and Marian Vittek. ELAN *V 3.4 User Manual.* LORIA, Nancy (France), fourth edition, December 1999.

[BDMN79] G.M. Birtwistle, O.-J. Dahl, B. Myhrhaug, and K. Nygaard. *Simula Begin.* Studenlitteratur, 1979.

[BKK+96] Peter Borovanský, Claude Kirchner, Hélène Kirchner, Pierre-Etienne Moreau, and Marian Vittek. ELAN: A logical framework based on computational systems. In José Meseguer, editor, *Proceedings of the first international workshop on rewriting logic*, volume 4 of *Electronic Notes in TCS*, Asilomar (California), September 1996.

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

[BKKR01] Peter Borovanský, Claude Kirchner, Helene Kirchner, and Christophe Ringeissen. Rewriting with strategies in ELAN: a functional semantics. *International Journal of Foundations of Computer Science*, 2001.

[Bor98] Peter Borovanský. *Le contrôle de la réécriture : étude et implantation d'un formalisme de stratégies.* PhD thesis, Université Henri Poincaré - Nancy I, 1998.

[Cas97] Guiseppe Castagna. *Object-Oriented Programming: A Unified Foundation.* Birkhauser, 1997.

[CDE+00] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martì-Oliet, J. Meseguer, and J.F. Quesada. Towards Maude 2.0. In Kokichi Futatsugi, editor, *WRLA2000, the 3rd International Workshop on Rewriting Logic and its Appli-*

cations, September 2000, Kanazawa, Japon. Electronic Notes in Theoretical Computer Science, 2000.

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

[CK99a] Horatiu Cirstea and Claude Kirchner. Combining higher-order and first-order computation using ρ-calculus: Towards a semantics of ELAN. In Dov Gabbay and Maarten de Rijke, editors, *Frontiers of Combining Systems 2*, Research Studies, ISBN 0863802524, pages 95–120. Wiley, 1999.

[CK99b] Horatiu Cirstea and Claude Kirchner. An introduction to the rewriting calculus. Research Report RR-3818, INRIA, 1999.

[CKL00] Horatiu Cirstea, Claude Kirchner, and Luigi Liquori. Matching power. Technical Report, LORIA, 2000.

[DK00a] Hubert Dubois and Hélène 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] Hubert Dubois and Hélène 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.

[DMT00] G. Denker, J. Meseguer, and C. Talcott. Formal Specification and Analysis of Active Networks and Communication Protocols: the Maude Experience. In *Proceedings DARPA Information Survivability Conference and Exposition DICEX 2000, Vol. 1, Hilton Head, South Carolina, January 2000*, IEEE, pages 251–265, 2000.

[FH86] F. Fages and G. Huet. Complete sets of unifiers and matchers in equational theories. *Theoretical Computer Science*, 43(1):189–200, 1986.

[Frü00] Thom Frühwirth. Proving Termination of Constraint Solver Programs. 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 298–317. Springer-Verlag, 2000.

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

[KB70] Donald E. Knuth and P. B. Bendix. Simple word problems in universal algebras. In J. Leech, editor, *Computational Problems in Abstract Algebra*, pages 263–297. Pergamon Press, Oxford, 1970.

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

[Rou88] M.-C. Rousset. On the Consistency of Knowledge Bases : The COVADIS System. In *Proceedings of the 8th European Conference on Artificial Intelligence*, pages 79–84, 1988.

[SS95] James G. Schmolze and Wayne Snyder. A Tool for Testing Confluence of Production Rule Systems. In M. Ayel and M.-C. Rousset, editors, *Proceedings of the European Symposium on the Validation and Verification of Knowledge-Based Systems. Université de Savoie, Chambéry, France*, 1995.

[ZN94] D. Zhang and D. Nguyen. PREPARE : A Tool for Knowledge Base Verification. *IEEE Transactions on Knowledge and Data Engineering*, 6(6):983–989, December 1994.