Rule-based programming and proving: the ELAN experience outcomes

Claude Kirchner, Hélène Kirchner

Rule-based programming and proving: the ELAN experience outcomes

Claude Kirchner and Hélène Kirchner

LORIA, CNRS & INRIA
615 rue du Jardin Botanique 54602 Villers-lès-Nancy, France
First.Last@loria.fr

All mgus are equal, but some mgus are more equal than others
Jean-Louis Lassez, Michael Maher and Kim Marriott [28]

Abstract. Together with the Protheo team in Nancy, we have developed in the last ten years the ELAN rule-based programming language and environment. This paper presents the context and outcomes of this research effort.

1 Introduction

Unification problems and their one-sided version *matching* are at the heart of schematic programming languages. These languages rely on pattern matching where patterns may be as different as first-order term or higher-order graphs or matrices; matching may be syntactic, equational or higher-order, and may identify the pattern and the subject using various forms of substitutions.

Because they offer a very high level of abstraction, many such languages have been designed and implemented to allow for high-level decision-making. We indeed assist currently to a main surge of interest in these languages from different point of views. First, in general purpose languages, matching capabilities are available for instance in ML or Haskell, while Prolog uses unification. Second, production systems and now business rule languages are fully based on pattern matching. They are prominent languages for higher-level decision-making and are central in expert systems and in companies prospective analysis. Third, the raising of XML, in particular as a general term-based syntax for the Web, allows the emergence of rewrite based languages like XSLT as well as for the so-called semantic web, where again languages like RuleML now emerge.

Indeed matching-based and more specifically rule-based languages have been prominent in algebraic specifications since at least twenty years. Several specification languages or programming environments, such as LARCH, OBJ, ASF+SDF [27], Maude [13], ELAN, to cite a few, are using term rewriting as their basic evaluation mechanism. From these experiences, we have inherited a deep knowledge somehow summarized in the CASL language [1].

In all the above mentioned languages and environments, rewriting and matching play an essential role, but few is done to give the user the possibility to control

the rewrite relation. Moreover, the semantics of some programming languages is imprecise with regard to the way rewritings are applied. One of the main originality of the ELAN project that we are going to detail in this paper, is to have pioneered strategic rewriting, i.e. strategy controlled rewriting.

This paper presents the main outcomes in terms of emerging concepts and lessons of this ten years experience of development of the ELAN language and practice. Several concepts presented in this paper are directly relevant to the context of high-level decision-making.

One of them is the conceptual difference between computation and deduction in a programming environment. Computations are described in the context of rewriting with normalizers, while deductions needs control, expressed by strategies. These two concepts are combined in ELAN to provide strategic rewriting. Another related outcome is the definition of a declarative strategy language that gives to the programmer the capability of precisely defining control.

Understanding and formalizing the concept of strategies in this context led to higher-order functionalities and to the rewriting calculus [11] which provides in particular a smooth integration of first-order rewriting and λ-calculus.

Providing strategic rewriting capability in existing programming language is a further step leading to pervasive rewriting and formal islands that are compiled into the hosting programming language.

This paper is organized as follows. The next section summarizes the main features of the ELAN language in a smooth way, explains and analyzes design choices and mentions missing capabilities. Section 3 enlights the main theoretical concepts that have emerged: strategic rewriting, strategy language, rewriting calculus, rewriting proof terms, pervasive rewriting. Then we share our experience on the practicality of strategic rewriting in Section 4. It illustrates by a few examples related to decision processes, the power of the strategic rewriting approach.

2 Main features of the language

The ELAN language is fully described in its user's manual [4] and at url `elan.loria.fr`. We can summarize its main characteristics by the "equation":

$$\text{ELAN} = \text{computation rules} + (\text{deduction rules} + \text{strategies})$$

The syntax of ELAN programs is given by a signature provided by the user and written in mixfix syntax. The semantics of the programs is given by computation and deduction rewrite rules together with strategies to control application of rules. ELAN programs are structured in modules, possibly parameterized and importing other modules.

Programming in ELAN is very easy when just computation is needed and can be quite elaborated when this is combined with powerful deduction and associated strategies. To program in ELAN, one should at least have an intuitive understanding of the two fundamental concepts of computation and deduction.

The important difference between them has been formally identified since one century by Henri Poincaré. Both concepts play a central role in today's proof theory as well as in semantics of programming languages. In proof theory, the status of what we search for and what needs to be computed should be identified and treated appropriately, in order to get proofs where only useful (and often difficult) parts are described. This is typical of *Deduction Modulo* as developed in [17]. In semantics because of its close relationship with computation as well as solving, in particular prominent in declarative programming languages.

Computation and deduction steps are defined in ELAN thanks to unlabeled and labeled rules respectively.

2.1 Computation

What is a computation? Here we call computation the normalization by a set of confluent and innermost terminating set of unlabeled rewrite rules.

The simplest kind of *rewrite rule* used in ELAN is an ordered pair of terms denoted $[]\ \ l \to r$ such that l, r are terms (we denote $\mathcal{T}(\mathcal{F}, \mathcal{X})$ the set of terms build over the signature $\mathcal{F}$ and the set of variables $\mathcal{X}$) and satisfying the usual restriction of their respective set of variables: $\mathcal{V}ar(r) \subseteq \mathcal{V}ar(l)$. The empty square brackets [] is important here, as it denotes the fact that this rule has no name.

A typical example of simple computation rules is the definition of addition on Peano numbers build using zero, denoted 0 and successor, denoted *succ*. The computation rules are

$$DEF(+) = \{[]\ \ x + 0 \to x \qquad []\ \ x + succ(y) \to succ(x + y)\}$$

They define, in particular in this simple example, a confluent and terminating rewrite system for which normal forms are the computation results in the tradition of algebraic specifications [2].

These rewrite rules define a *congruence* on the set of terms, therefore they are potentially applied everywhere in a term: at the root as well as at any occurrence. The built-in strategy used in ELAN to implement the normalization process by these simple rewrite rules is *left-most inner-most* and the ELAN compiler is able to apply more than 15 millions of such rewrite steps per second. These "simple" rules are already extremely powerful since a single left-linear and even regular rewrite rule is enough to be Turing complete (but of course in this case non-terminating!) [15].

The behavior of these rewrite rules is simple as their application is decided locally, independently of their application context, and could implicitly involve an equality check when they are non-linear. Such rule are at the heart of many algebraic languages like OBJ [21], ASF [34] or LPG [3]. When one wants to add the possibility to specify contextual information, condition are added to simple rules, and they are denoted in ELAN: $[]\ \ l \to r$ **if** c, where c is a boolean expression.

The next extension of the conditional rewrite rule provided by ELAN, consists in adding the capability to control not only the application context, but also the

way computation are done. This is provided through different features. First, ELAN provides the capability to share some results. This is done by generalized conditional rewrite rules of the form $[\,]\ \ l \to r$ **where** $p_1 := c_1, \ldots,$ **where** $p_n := c_n$ whose behavior consists of (1) searching in the term t to be evaluated a subterm at occurrence ω that matches l with a substitution σ (i.e. $\sigma(l) = t_{|\omega}$), then (2) finding a match σ_1 from p_1 to $\sigma(c_1)$ (i.e. $\sigma_1(p_1) = \sigma(c)$),..., then $(n+1)$ find a match σ_n from p_n to $\sigma_{n-1} \ldots \sigma_1\sigma(c_1)$ (i.e. $\sigma_n(p_n) = \sigma_{n-1} \ldots \sigma_1\sigma(c)$) and finally replace $t_{|\omega}$ by $\sigma_n\sigma_{n-1} \ldots \sigma_1\sigma(r)$.

This more technical explanation can be easily understood using the following example. Consider an operation $doubleflat$ on lists, that takes a list (for instance ((1.2).(3.4))) and builds the concatenation of its flattened form with the reverse of its flattened form (in this case (1.2.3.4.4.3.2.1)) can be defined by the rule with two matching conditions:

$$[\,]\ doubleflat(l) \to append(x, y) \ \textbf{where}\ x := flatten(l) \ \textbf{where}\ y := reverse(x)$$

The interest of this form with respect to the classical rule

$$[\,]\quad doubleflat(l)\ \to\ append(flatten(l), reverse(flatten(l)))$$

is indeed to factorise the expression of $flatten(l)$ giving to the programmer the possibility to avoid computing twice the flattened form of l.

Using unlabeled rewrite rules, ELAN provides the programmer with the ability to define *normalizers* i.e. functions that return the unique normal form. This is quite powerful but sometimes surprising for the programmer as well as the end-user: in the above example on Peano arithmetic, the term $succ(0) + succ(0)$ will never be accessible, only the term $succ(succ(0))$ will be visible.

2.2 Strategic rewriting

As already said, rewrite rules are natural to express computation but also deduction, i.e. rewrite systems that are neither necessarily confluent nor terminating.

While keeping the control over the evaluation of confluent and terminating rewrite rules is not essential (even through it could be useful for describing an efficient way to reach the normal form), it is mandatory for either non-terminating or non-confluent systems.

To provide the control on the execution order of a rewrite rule system, we use in ELAN a non empty rule label. These labels have indeed two purposes: first they signal that the rule is a deduction rule, second they provide a name that will be useful when describing elaborated strategies. A simple labeled rule is therefore of the form $[\ell]\ \ l\ \to\ r$ where ℓ is a non-empty label, l and r are terms which variables satisfy as usual $\mathcal{V}ar(r) \subseteq \mathcal{V}ar(l)$. Typical simple examples of such labeled rewrite rules are (assuming x, y to be variables and a a constant) [`id`] $x\ \to\ x$ or [`constant-a`] $x\ \to\ a$ or [`sum`] $x + succ(y) \to succ(x + y)$ or [`proj1`] $x + y\ \to\ x$ or [`proj2`] $x + y\ \to\ y$. Notice that an equational interpretation of such rules is in general useless.

To make a clear cut between the normalizers —that implicitly embed a traversal strategy— and the deduction rules, the application of a labeled rewrite rule is performed *only* at the top most occurrence of the term on which it is applied and this application consumes the rewrite rule. This is totally similar to a function application in functional programming. For example, the application of `id` to any term t returns in one step t, and the application of `constant-a` to t terminates and returns in one step a. But, the application of `sum` fails on $0 + (0 + succ(0))$, since the redex is not at the top occurrence.

At the current state of our description, we can define a *primal* strategy as a labeled rewrite rule. Applying such a strategy $\mathcal{S}$ on a term t is denoted $\mathcal{S}(t)$. When l matches t (i.e. $\exists \sigma, s.t.\ \sigma(l) = t$), the application result is $\sigma(r)$ and we say that the strategy succeeds. When l does not match t, we say that the strategy fails and the application result is the empty set $\emptyset$. Indeed we will see below that in general the application of a strategy to a term, when it terminates, is a finite (ordered) multiset of terms (i.e. a flat list).

Two strategies can be concatenated by the symbol ";", i.e. the second strategy is applied on all results of the first one. $\mathcal{S}_1; \mathcal{S}_2$ denotes the sequential composition of the two strategies. It fails if either $\mathcal{S}_1$ fails or $\mathcal{S}_2$ fails on all results of $\mathcal{S}_1$ and in this case the application result is $\emptyset$. Otherwise, its results are all results of S_1 on which S_2 is successfully applied.

The next natural strategy combinator used in ELAN is dk. It takes a list of strategies and $\mathsf{dk}(\mathcal{S}_1, \ldots, \mathcal{S}_n)$ applies all strategies and for each of them returns all its results. Its application result is the union of all the application results of the individual strategies $\mathcal{S}_i$. It may be empty, in which case we say that the strategy fails.

Together with the non-determinism capability provided by the dk operator, the analog of a cut operation is provided par the first_one strategy constructor. $\mathsf{first\_one}(\mathcal{S}_1, \ldots, \mathcal{S}_n)$ chooses the first strategy $\mathcal{S}_i$ in the list that does not fail, and returns one of its first results. This strategy returns at most one result or fails if all sub-strategies fail.

Iterators are also provided: for example, $\mathsf{repeat*}(\mathcal{S})$ applies repeatedly the strategy $\mathcal{S}$ until it fails and returns the results of the last unfailing application. This strategy can never fail (zero application of $\mathcal{S}$ is always possible) and may return more than one result.

The full description of strategy combinators available in ELAN can be found in [6,5]. This simple language allows us to build more elaborated strategies and a rewrite rule is a natural way to give a name to such an expression like in [] `simpleStrat` $\rightarrow$ $\mathsf{dk}($`id`, `constant-a`$)$. But notice that here, we are not only rewriting terms but strategy expressions.

We are ready to define the general form of an ELAN rewrite rule: A *labeled rewrite rule with general matching conditions* is denoted

$$[\ell]\ \ l \rightarrow r \ \textbf{where}\ p_1 := \mathcal{S}_1(u_1), \ldots, \textbf{where}\ p_n := \mathcal{S}_n(u_n)$$

and $l, r, p_i, u_i \in \mathcal{T}(\mathcal{F}, \mathcal{X})$, the $\mathcal{S}_i$ are strategy expressions and a variable is used somewhere only if it is "well defined", a natural technical condition detailed in

the user's manual. The relation induced on terms by these rules is called *strategic rewriting*.

The application result of such a rule on a term t is defined as follows: (1) match l against t using a substitution σ, then (2) match p_1 against all the results of the application of $\mathcal{S}_1$ on $\sigma(u_1)$. Let σ_1^i be such a match, (3) match p_2 against $\sigma_1^i\sigma(u_2)$, ... and finally the result consists of the multiset of all the instances of r computed in the **where** part.

When the label of the above rule is empty, the application process is performed everywhere in the term t as for simple unlabeled rules.

From the basic strategy combinators provided by the language, the user can define his own ones, such as in the following strategy expression used in Colette [9] to describe and solve CSP:

```
[] FLAChoicePointSplitLastToFirstAll =>
dk (LocalConsistencyForECandDC);
repeat* (
    dk (first one (SplitDomainSecondMiddle),
        first one (SplitDomainFirstMiddle));
    first one (first one (ExtractConstraintsOnEqualityVar);
               first one (Elimination, id);
               LocalConsistencyForEC
               ,
               first one (ExtractConstraintsOnDomainVar);
               LocalConsistencyForEC
               ,
               id)
         );
first one (GetSolutionCSP)
```

2.3 Summing up the main language design choices

In ELAN, computation and deduction are modeled using unlabeled and labeled rules respectively. The control over labeled rules is performed using strategies described by a simple strategy language where iterators and non-determinism operators are available.

The matching process underlying all operations in ELAN has been designed to be first-order, therefore we do not allow higher-order variables in rewrite rules. This is a strong design decision relying on the facts that, if needed, higher-order can be encoded at the first-order level (see for example [16]) and second, that first-order interpretation and compilation techniques are much better understood than for higher-order (where indeed at order five, matching decidability is still an open problem for beta-eta conversion [29]).

Since the strategy language allows non-determinism, the implementation choices have been between returning explicit multisets of results or enumerating these results using backtracking. Even though extensions of the language support the "set-of" capability for returning multisets of results, the basic evaluation mechanism relies on backtracking.

2.4 Added and missing capabilities

The design of a programming language is a subtil blend of conceptual choices about its main characteristics with several additional non essential but useful capabilities, altogether carefully and smartly implemented. We give here our view of the goodies and missing of the current ELAN language.

Goodies Indeed, ELAN comes with many goodies that make it more attractive and usable as a practical programming language.

Pre-processing Since higher-order matching (and in particular second-order) is not directly available, a pre-processor is quite convenient to overcome this limitation. A typical example is for syntactic unification, where the decomposition rule which indeed depends on the signature, could be expressed as follows:

```
FOR EACH SS:pair[identifier,int]; F:identifier;  N:int
 SUCH THAT SS:=(listExtract) elem(Fss)
       AND  F:=()first(SS) AND N:=()second(SS) :{
  rules for unifPb
     s_1,...,s_N:term; t_1,...,t_N:term;
  local
  [decompose] P ^ F(s_1,...,s_N)=F(t_1,...,t_N) => P { ^ s_I=t_I }_I=1...N
  end
 end  }
```

If the signature contains two symbols, g of arity 1 and f of arity 2, then the pre-processor will generate the two rules:

```
[decompose] P ^ g(s1)=g(t1)       => P ^ s1 = t1 end
[decompose] P ^ f(s1,s2)=f(t1,t2) => P ^ s1 = t1 ^ s2 = t2 end
```

Earley parsing In the tradition of several algebraic programming languages like OBJ, ELAN provides the user with the capability to define its own mixfix syntax in a very liberal way. It is then analyzed using Earley's algorithm. This is quite convenient to adapt the syntax to the user's description of the problem, at the price of a cubic parsing process, in the worst case.

Modularity and parameterization A serious programming language cannot come without modularization capabilities. ELAN provides the possibility to define local and non-local operators and rules, to import modules and to defined parameterized modules.

Rewriting modulo AC An important improvement in the rewriting agility is the capability to take into account, at run time, properties of operators like associativity and commutativity. This frees the programmer of tediously describing all the possible variations in rule application when one of its symbol is associative and commutative (AC for short) but AC matching is not available. The code

reduces significantly at the price of a matching algorithm running in exponential time in the worst case. A clever compiler have been designed to keep this drawback as much under control as possible [26].

Missing A language cannot contain every fashionable features. We have made some design choices but some goodies could have been retrospectively usefully introduced. This section sums up our current views.

Enriched signatures The current system is based on many-sorted signatures. The agility of the language could have been enhanced by using order-sorted or membership constraints [33]. But the implementation becomes much more intricated, in particular since the sort of a term may in this case change at run time. Secondly, the interaction of the matching theories and the extended sort capabilities can be quite subtle, even for experienced programmers. Therefore we did not provide this feature as it is indeed done in Maude [13].

Traversals ELAN provides elaborated strategies as we have seen before, but term traversals are not provided. This is due to the fact that the concept emerged after the main design of the language was achieved [35]. Traversals are useful and completely in line with the language design. Even if some early versions of the language prototyped them, they are not provided in the current distribution and could have been a useful extension.

Matching theories The debate on which matching theories are useful and could be usefully implemented has been (and is still!) long-standing in the ELAN team. On one side, one can say that the more the better, on the other hand, the more theory the most complex the implementation becomes. Moreover, when combining several theories, completeness of matching becomes a real difficulty, in particular on the complexity side. Therefore ELAN provides only syntactic and AC matching and their combination. Adding associativity is natural and useful: it is indeed the first theory available in TOM (see Section 3.4).

Deduction modulo relies on the ability to embed potentially complex theories in the modulo part, and therefore in the matching process. It would be useful to give the possibility to match modulo user-defined theories: a capability far beyond the currently available knowledge and rewrite technology if we want to keep the system efficient.

Deep inference The design decision to have unlabeled rules applied everywhere (acting as normalizers) and labeled ones to be applied only on top of term (acting as deduction rules), comes from the usual view of deduction rule applied at the top level of formulas as in the calculus of sequent or in natural deduction. Recent works on deep inference [8] show all the interest to have inference rules also applied inside terms or formulas. ELAN is not designed this way, but this is certainly a challenging capability to have deep inference available, for example via a clever combination of labeled rule with traversals.

3 Emerging concepts

The general setup, design and implementation of the system leaded the ELAN team to elaborate or refine several main ideas and results. We review here the main emerging concepts.

3.1 Semantics of strategic rewriting

One of the main originality and useful feature of ELAN is the ability to define and efficiently execute rewriting strategies.

The concern of giving to the programmer some control on the normalization process is already present in OBJ with the so-called local strategies. Further extension of this idea is also related to the control of concurrent evaluation [22]. On the proof side, strategies are called tactics or plans and are mandatory in proof assistants.

What ELAN brings first here is that term rewriting is used for both computation (and therefore normalization) and deduction. This means that rewriting is used to model both equality and transition. As a consequence strategies are not only a (useful) addition but a mandatory one.

However in order to understand the semantics of strategic rewriting, one needs to understand the concept of strategy. It is indeed very simple and natural to define a *strategy as a set of proof terms of a rewrite theory* as proposed in [25, 36] using the rewriting logic [30] framework.

3.2 Strategy language

Clearly, an arbitrary set of proof terms may be very elaborated or irregular from the computational point of view, it could be in particular non-recursive. This is why languages describing special subclasses of strategies are needed. The ELAN strategy constructors described in Section 2.2 contribute to define such a subclass.

However this is not expressive enough to allow recursive and parameterized strategies. This is why the more general notion of defined strategies has been introduced [5, 6]. Their definition is given by a strategy operator with a rank, and a set of labeled rewrite rules. The example of a map functor on lists illustrates this.

Let map be of rank map : $(\langle s \mapsto s \rangle)\ \langle list[s] \mapsto list[s] \rangle$ where $\langle s \mapsto s \rangle$ and $\langle list[s] \mapsto list[s] \rangle$ are strategy sorts. The argument of map must be a strategy S that applies to a term of sort s and returns results of sort s. The strategy $\mathsf{map}(S)$ applies to a term of sort $list[s]$ and returns results of sort $list[s]$. It is defined by the rewrite rule:

$$[\,]\ \ \mathsf{map}(S) \rightarrow \mathsf{first}(\mathsf{nil}, S \cdot \mathsf{map}(S)) \qquad (1)$$

where S is a variable of sort $\langle s \mapsto s \rangle$. The right-hand side of this definition means that whenever the strategy $\mathsf{map}(S)$ is applied to a term t, either t is nil, or the

strategy S is applied to the head of t (i.e. t should be a non-empty list) and $\mathsf{map}(S)$ is further applied to the tail of t.

Allowing the description of a strategy by a set of rewrite rules, as above, strongly increases the expressive power of ELAN: strategies may be recursive, parameterized and typed as well. But this also leads to understand a strategy as a function, or later on as a functional. Also adding an explicit application operator leads to the formalization of higher-order objects. This gave rise to the rewriting calculus which provides both a language to express strategies and an operational semantics for the constructions of ELAN.

3.3 The rewriting calculus

The rewriting calculus, whose initial design is detailed in [11], is also called ρ-calculus. It is a natural but prominent outcome of the ELAN design, implementation and usage. As we have seen in the previous sections, rewriting strategies are central in ELAN and this leaded to the simple idea that the simplest strategy is indeed just a rule (i.e. $l \twoheadrightarrow r$) and that applying this strategy on a term t is just (explicitly) applying this rule, i.e. $(l \twoheadrightarrow r \;\; t)$. Pushing this remark further leaded to the first main idea of the calculus, that is to provide a uniform combination of term rewriting and lambda-calculus. This is in particular fully adapted to the description of computation and deduction transitions.

The second main idea of the calculus is to make all basic ingredients of rewriting explicit objects, in particular the notions of rule *formation*, *application* and *result*. Terms, rules, rule application and therefore rule application strategies are all treated as first class objects. To make this more explicit, the rule constructor "$\twoheadrightarrow$" uses a different arrow symbol in all this Section. For example, using a syntax close those of lambda-calculus, application of the rule $2 \twoheadrightarrow s(s(0))$, to a term, e.g. the constant 2, is explicitly represented as the object $(2 \twoheadrightarrow s(s(0)) \;\; 2)$ which evaluates to $s(s(0))$.

The third important concept in the ρ-calculus is that rules are fired modulo some theory e.g. associativity and commutativity. For example, provided the commutativity of $+$, the ρ-term $(x + 0 \twoheadrightarrow x \;\; 0 + 1)$ reduces to 1. The fact that results are explicit allows us to give a precise meaning to the reduction of the ρ-term $(x + y \twoheadrightarrow x \;\; a + b)$ as the structure $a \,\mathbf{|}\, b$ that may be informal understood as a set of results.

Since the beginning, we wanted to integrate explicit substitutions [10] but the link with matching constraints has been done later [12] and a combined version is presented in [23]. Indeed to represent explicitly substitution is useful not only from the foundational point of view, but also for representing proof term, as we will see later.

As usual, for a calculus with binder, we work modulo the *α-conversion* and adopt Barendregt's *hygiene-convention i.e.* free and bound variables have different names. The syntax is the following:

$$\begin{array}{lcll} \mathcal{P} & ::= & \mathcal{T} & \text{Patterns} \\ \mathcal{T} & ::= & \mathcal{X} \mid \mathcal{K} \mid \mathcal{P} \twoheadrightarrow \mathcal{T} \mid \mathcal{T} \;\; \mathcal{T} \mid [\mathcal{P} \ll \mathcal{T}]\mathcal{T} \mid \mathcal{T} \,\mathbf{|}\, \mathcal{T} & \text{Terms} \end{array}$$

1. $A \twoheadrightarrow B$ denotes a *rule abstraction* with pattern A and body B; the free variables of A are bound in B.
2. $(A\ \ B)$ denotes the *application* of A to B.
3. $[P \ll A]B$ denotes a *delayed matching constraint* with pattern P, body B and argument A; the free variables of P are bound in B but not in A.

To obtain good properties for the calculus (*e.g.* confluence) the form of patterns has to be restricted to particular classes of ρ-terms. The operator $[\_ \ll \_]\_$ can be decorated by (matching) theory $\mathbb{T}$ and become $[\_\ll_{\mathbb{T}}\_]\_$ if this is not implicit when working in a given context. The set of solutions of the matching constraint $[P\ll_{\mathbb{T}} B]A$ is denoted $\mathcal{S}ol(P\prec_{\mathbb{T}} B)$.

The *small step semantics* of the ρ-calculus is given by the following rules:

$$
\begin{array}{lll}
[\rho] & (P \twoheadrightarrow A\ \ B) & \rightarrow_{\rho}\ [P\ll_{\mathbb{T}} B]A \\
[\sigma] & [P\ll_{\mathbb{T}} B]A & \rightarrow_{\sigma}\ A\theta_1,\ldots,A\theta_n \quad \text{with } \{\theta_1,\ldots,\theta_n\} = \mathcal{S}ol(P\prec_{\mathbb{T}} B) \\
[\delta] & (A \,\mathbf{|}\, B\ \ C) & \rightarrow_{\delta}\ (A\ \ C)\,\mathbf{|}\,(B\ \ C)
\end{array}
$$

For example, the β-redex $(\lambda x.t\ u)$ is nothing else than the ρ-redex $(x \twoheadrightarrow t\ \ u)$ (i.e., the application of the rewrite rule $x \twoheadrightarrow t$ to the term u) which reduces to $[x \ll u]t$ and then to $\{x/u\}t$ (i.e., the application of the higher-order substitution $\{x/u\}$ to the term t).

The small-step semantics can then be customized, via the rule $[\sigma]$, in order to consider non-unitary and even infinitary theories [12].

These ρ-calculus principles, that can also be understood as a kind of constrained rewriting, emerged from the ELAN design: they soon attracted a lot of attention and have been studied for themselves (see the latest developments at url `rho.loria.fr`). For example, a version of the rewriting calculus with explicit substitutions has been used to represent rewriting derivations modulo AC [32] for the Coq proof assistant.

3.4 TOM and formal islands

Since the beginning of the ELAN project, we have been strongly concerned with the feasability of strategic rewriting as a practical programming paradigm. Therefore, the development of efficient compilation concepts and techniques took an important place in the language support design. The results presented in [26] leaded to a quite efficient implementation and thus demonstrated the practicality of the paradigm.

But even if ELAN is a nice language cleverly and efficiently implemented, it requires an existing application to be totaly rewritten in ELAN to benefit from its capabilities. Strategic rewriting is therefore available but hardly usable in the large.

This is the main concern that leaded to the emergence of the idea of *formal island*, a general way to make formal methods, and in particular matching and rewriting available in virtualy any existing environment. TOM [31] is an

implementation of this idea. In its Java instance, TOM provides matching and rewriting primitives that are added to the Java language. These specific instructions are then compiled to the host language (e.g. Java), using similar techniques as those used for compiling ELAN. The good things are that one can then use the normal forms provided by rewriting to get conciseness and expressiveness in Java programs, but moreover one can prove that these sets of rewrite rules have useful properties like termination or confluence. Once the programmer has used rewriting to specify functionalities and to prove properties, the compiled dissolves this formal island in the existing code just by compilation. The use of rewriting and TOM therefore induces no dependence: once compiled, a TOM program contains no more trace of the rewriting and matching statements that were used to build it.

TOM and its Eclipse environment are available at url `tom.loria.fr`: they provide an efficient way to integrate rewriting in Java as well as to perform easy and formaly safe XML rewriting.

4 Applications to high-level decision making

How does this research contribute to high-level decision making? The interested reader may consult the ELAN web page to get an exhaustive idea of developped applications, but we choose here to select four of them to illustrate how ELAN and strategic programming can be used to model different kinds of processes, namely solving, computing and proving.

4.1 Constraint solving

One of the first concern of the ELAN project was to model constraint solving, from unification problems to complex Constraint Satisfaction Problems (CSP). Research has been very active on CSP since the seventies and often in relationship with traditional Operational Research techniques and Constraint Logic Programming, in particular with the seminal works on Prolog [14] and CLP [24]. Our concern on this topics was to express in a simple and clear way the underlying concepts used to solve CSP, formalized as a deduction process. The ELAN language is especially well-suited, thanks to the explicit definition of deduction rules and control. Actions are associated with rewrite rules and control with strategies that establish the order of applications of deductions. Expressing the algorithms developed for solving CSP as rewrite rules driven by strategies leads to a better understanding, easy combination and potentially to improvements and proof of correctness. Colette [9] is a CSP solving environment implemented in ELAN to validate this approach. Various searching techniques for clever exploration of the solutions space, problem reduction techniques that transform a CSP into an equivalent problem by reducing the values that the variables can take, as well as various forms of consistency algorithms are described by ELAN strategies.

4.2 Chemical computations

Rule-based systems and strategies have been used for modelling a complex problem of chemical kinetic: the automated generation of reaction mechanisms. The generation of detailed kinetic mechanisms for the combustion of a mixture of organic compounds in a large temperature field requires to consider several hundred chemical species and several thousands of elementary reactions. An automated procedure is the only convenient and rigorous way to write such large mechanisms. Flexibility is often absent or limited to menu systems, whereas the actual use of these systems, during validation of generated mechanisms by chemists, as well as during their final use for conception of industrial chemical processes, requires modifications, activations or deactivations of involved rules according to new experimental data, reactor conditions, or chemist expertise. The purpose of an automated generator of detailed kinetic mechanisms is to take as input one or more hydrocarbon molecules and the reaction conditions and to give as output the list of elementary reactions applied and the corresponding thermodynamic and kinetic data. The GasEl[7] system has been designed in ELAN for that purpose. The representation of the chemical species uses the notion of molecular graphs, encoded by a term structure called GasEl terms. The chemical reactions are expressed by rewrite rules on molecular graphs, encoded by a set of conditional rewrite rules on GasEl terms. ELAN's strategy language is quite appropriate to express the reactions chaining in the mechanism generator. The required flexibility is provided by the high-level specification of the declarative strategy language, that can reflect the chemist's decisions.

4.3 Proving

Several proof tools have been designed in ELAN, ranging from a predicate logic prover, or a completion procedure, to various model checkers. In the context of rule-based programming, termination is a key property that warrants the existence of a result for every evaluation of a program. CARIBOO is a termination proof tool for rewrite programs, given by sets of rewrite rules. Its foundation is a termination proof method based on an explicit induction mechanism on the termination property. CARIBOO is able to deal with different term traversal strategies, corresponding to call-by-value (innermost strategy [19]), call-by-name (outermost strategy [20]), or more local calls (local strategies on the operators [18]). Such proof tools might be later used in proof environments able to combine them in order to guarantee safety of programs. The interesting point here is that the proof procedure is described in each case by deduction rules and a control specification, which are directly reflected in the ELAN implementation.

4.4 Combining computation and deduction

The rewriting calculus is quite useful in combining rewriting-based automated theorem proving and user-guided proof development, with the strong constraint

of safe cooperation of both. We addressed this problem in practice in combining the Coq proof assistant and the ELAN rewriting based system.

The approach followed for equational proofs relies on a normalization tactic in associative and commutative theories written in ELAN. It generates a proof term in the rewriting calculus, which is then translated into a proof term written in the calculus of constructions syntax that can finally be checked by Coq to get the proof of the normalization process [32]. The advantages of this approach are to take benefit from the efficient (conditional AC) rewriting performed by the ELAN compiler, and to ease the size reducing transformations of the proof terms before sending them to Coq. For that, the ELAN compiler has been extended by a proof term producer that builds the rewriting proof term, and by a proof term translator that transforms this formal trace of ELAN into the corresponding Coq proof term for checking. In this cooperation scheme, ELAN can be seen as a computing server and Coq proof sessions as its clients.

Actually this work goes beyond the specific use of Coq and ELAN. It raises the general problem of incorporating decision procedures in proof assistants based on type theory, in a reliable and efficient way. Reliability is handled here through the concept of proof term, that contains all information about the proof and is exchanged between the two systems. Built by ELAN during the rewriting proof construction, it is then checked by Coq or by any proof assistant.

4.5 When is ELAN not appropriate?

While rules and strategies are really natural in defining normalizers or when having to model deductive or transition systems, some applications are not easy to program in the current version of ELAN.

In a first place, since ELAN is a *term* rewrite rule language, data structures such as graphs or matrices, are not easy to deal with. When working with such structures, encodings, sometimes clever like in the chemistry application reported above, are necessary to take benefit of the language features.

We did not develop any elaborated numerical computations and types nor fancy input/output and graphical libraries. This means that connecting ELAN programs with the outside world is possible but not easy. Any application heavily relying on such characteristics is not currently appropriate to develop in the language.

In the context of ambitious applications, it is fundamental to interface ELAN or its fundamental concepts of rules and strategies with other programming languages. This is the place where TOM comes into play and is already quite promising.

5 Conclusion

One picture is better than hundred explanations: this popular sentence summarizes the interest to deal with modeling environments making a fundamental use

of patterns. The research community in informatics is rich in such languages where Prolog, CLP, and rule-based programming play a central role.

Our researches on these topics for the last ten years has been very fruitful, both in practical terms as well as in fundamental research advances: among them, the demonstration that rewrite-based languages could be as efficient as functional ones, the emergence of the rewriting calculus as a unified environment for strategic rewriting and functional evaluation, and the explicitation of the computation and deduction concepts as a consistent programming paradigm.

Because of the pattern matching capability of the language, the applications developped in the ELAN language have shown that programming an algorithm, solving a constraint or searching for a proof are of the very same essence and, indeed, are in general collaborative tasks for which rule-based programming is very well adapted.

As the programming activity is abstracting more and more to allow for higher-level decision-making, rule-based programming offers a clever and useful paradigm that becomes more and more attractive. We hope that the ELAN experience outcomes will contribute to make it more useful and popular .

Acknowledgements

Our sincere and warmest thanks go first to the past and present members of the Protheo team who contributed to the ELAN project, either as designers, users or contributors to theoretical or practical problems.

Such a ten years project could not fruifully happen without the strong support of our host institutions: INRIA, CNRS and the Universities of Nancy.

Our interactions with thematically related groups have been quite fruitful and we are very pleased to thank the SRI International team leaded by Joe Goguen, José Meseguer and Carolyn Talcott and from which emerged OBJ and Maude, the CWI team leaded by Paul Klint and Mark van den Brand who are developping ASF+SDF and who share with us the successful Aircube project.

Our initial interest for rewriting and its applications was triggered and driven by Jean-Pierre Jouannaud. His long support and interest as well as the interactions with his teams have always been particularly fruitful.

Jean-Louis Lassez has been at the start of this project in 1993 for the first presentation of the concepts and system: it is our great pleasure to dedicate him this paper.

References

1. E. Astesiano, M. Bidoit, H. Kirchner, B. Krieg-Br?ckner, P. D. Mosses, D. Sannella, and A. Tarlecki. CASL: The Common Algebraic Specification Language. *Theoretical Computer Science*, 286(2):153–196, September 2002.
2. J. A. Bergstra and J. V. Tucker. Algebraic specifications of computable and semicomputable data structures. *Theoretical Computer Science*, page 24, 1983.

3. D. Bert, P. Drabik, R. Echahed, O. Declerfayt, B. Demeuse, P.-Y. Schobbens, and F. Wautier. Reference manual of the specification language LPG- version 1.8 on SUN workstations. Internal report, LIFIA-Grenoble (FRANCE), January 1989.
4. P. Borovanský, H. Cirstea, H. Dubois, C. Kirchner, H. Kirchner, P.-E. Moreau, Q.-H. Nguyen, C. Ringeissen, and M. Vittek. ELAN *V 3.6 User Manual*. LORIA, Nancy (France), fifth edition, February 2004.
5. P. Borovansky, C. Kirchner, H. Kirchner, and P.-E. Moreau. ELAN from a rewriting logic point of view. *Theoretical Computer Science*, 2(285):155–185, July 2002.
6. P. Borovanský, C. Kirchner, H. Kirchner, and C. Ringeissen. Rewriting with strategies in ELAN: a functional semantics. *International Journal of Foundations of Computer Science*, 12(1):69–98, February 2001.
7. O. Bournez, G.-M. Côme, V. Conraud, H. Kirchner, and L. Ibănescu. A rule-based approach for automated generation of kinetic chemical mechanisms. In *Proceedings 14th Conference on Rewriting Techniques and Applications, Valencia (Spain)*, Lecture Notes in Computer Science. Springer-Verlag, 2003.
8. K. Brünnler. *Deep Inference and Symmetry in Classical Proofs*. Logos Verlag, Berlin, 2004.
9. C. Castro. Building Constraint Satisfaction Problem Solvers Using Rewrite Rules and Strategies. *Fundamenta Informaticae*, 34:263–293, September 1998.
10. H. Cirstea. *Calcul de réécriture : fondements et applications*. Thèse de Doctorat d'Université, Université Henri Poincaré - Nancy I, 2000.
11. H. Cirstea and C. Kirchner. The rewriting calculus — Part I *and* II. *Logic Journal of the Interest Group in Pure and Applied Logics*, 9(3):427–498, May 2001.
12. H. Cirstea, C. Kirchner, and L. Liquori. Matching Power. In A. Middeldorp, editor, *Rewriting Techniques and Applications*, volume 2051 of *Lecture Notes in Computer Science*, Utrecht, The Netherlands, May 2001. Springer-Verlag.
13. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. F. Quesada. Maude: Specification and programming in rewriting logic. *Theoretical Computer Science*, 2(285), 2001.
14. A. Colmerauer. An introduction to Prolog III. *Communications of the ACM*, 33(7):69–90, 1990.
15. M. Dauchet. Simulation of Turing machines by a left-linear rewrite rule. In N. Dershowitz, editor, *Proceedings 3rd Conference on Rewriting Techniques and Applications, Chapel Hill (N.C., USA)*, volume 355 of *Lecture Notes in Computer Science*, pages 109–120, April 1989.
16. G. Dowek, T. Hardin, and C. Kirchner. HOL-$\lambda\sigma$ an intentional first-order expression of higher-order logic. *Mathematical Structures in Computer Science*, 11(1):21–45, 2001.
17. G. Dowek, T. Hardin, and C. Kirchner. Theorem proving modulo. *Journal of Automated Reasoning*, 31(1):33–72, Nov 2003.
18. O. Fissore, I. Gnaedig, and H. Kirchner. Termination of rewriting with local strategies. In M. P. Bonacina and B. Gramlich, editors, *Selected papers of the 4th International Workshop on Strategies in Automated Deduction*, volume 58 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers B. V. (North-Holland), 2001.
19. O. Fissore, I. Gnaedig, and H. Kirchner. CARIBOO : An induction based proof tool for termination with strategies. In *Proceedings of the Fourth International Conference on Principles and Practice of Declarative Programming*, pages 62–73, Pittsburgh (USA), October 2002. ACM Press.

20. O. Fissore, I. Gnaedig, and H. Kirchner. Outermost ground termination. In *Proceedings of the Fourth International Workshop on Rewriting Logic and Its Applications*, volume 71 of *Electronic Notes in Theoretical Computer Science*, Pisa, Italy, September 2002. Elsevier Science Publishers B. V. (North-Holland).
21. J. A. Goguen. Some design principles and theory for OBJ-0, a language for expressing and executing algebraic specifications of programs. In E. Blum, M. Paul, and S. Takasu, editors, *Mathematical Studies of Information Processing*, volume 75 of *Lecture Notes in Computer Science*, pages 425–473. Kyoto (Japan), 1979. Proceedings of a Workshop held August 1978.
22. J. A. Goguen, C. Kirchner, and J. Meseguer. Concurrent term rewriting as a model of computation. In R. Keller and J. Fasel, editors, *Proceedings of Graph Reduction Workshop*, volume 279 of *Lecture Notes in Computer Science*, pages 53–93, Santa Fe (NM, USA), 1987.
23. Horatiu Cirstea, Germain Faure, and Claude Kirchner. A rho-calculus of explicit constraint application. In *Proceedings of the 5th workshop on rewriting logic and applications*. Electronic Notes in Theoretical Computer Science, 2004.
24. J. Jaffar and J.-L. Lassez. Constraint logic programming. In *Proceedings of the 14th Annual ACM Symposium on Principles Of Programming Languages, Munich (Germany)*, pages 111–119, 1987.
25. C. Kirchner, H. Kirchner, and M. Vittek. Designing Constraint Logic Programming Languages using Computational Systems. In P. Van Hentenryck and V. Saraswat, editors, *Principles and Practice of Constraint Programming. The Newport Papers.*, pages 131–158. MIT press, 1995.
26. H. Kirchner and P.-E. Moreau. Promoting rewriting to a programming language: A compiler for non-deterministic rewrite programs in associative-commutative theories. *Journal of Functional Programming*, 11(2):207–251, 2001.
27. P. Klint. A meta-environment for generating programming environments. *ACM Transactions on Software Engineering and Methodology*, 2:176–201, 1993.
28. J.-L. Lassez, M. J. Maher, and K. Marriot. Unification revisited. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*. Morgan-Kaufman, 1988.
29. R. Loader. Higher order β matching is undecidable. *Logic Journal of the IGPL*, 11(1):51–68, 2003.
30. J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96:73–155, 1992.
31. P.-E. Moreau, C. Ringeissen, and M. Vittek. A Pattern Matching Compiler for Multiple Target Languages. In G. Hedin, editor, *12th Conference on Compiler Construction, Warsaw (Poland)*, volume 2622 of *LNCS*, pages 61–76. Springer-Verlag, May 2003.
32. Q.-H. Nguyen, C. Kirchner, and H. Kirchner. External rewriting for skeptical proof assistants. *Journal of Automated Reasoning*, 29(3-4):309–336, 2002.
33. G. Smolka, editor. *Special issue on Order-sorted Rewriting*, volume 25 of *Journal of Symbolic Computation*. Academic Press inc., April 1998.
34. A. van Deursen, J. Heering, and P. Klint. *Language Prototyping*. World Scientific, 1996. ISBN 981-02-2732-9.
35. E. Visser. Language independent traversals for program transformation. Technical report, Department of Computer Science, Universiteit Utrecht, Utrecht, The Netherlands, 2000.
36. M. Vittek. *ELAN: Un cadre logique pour le prototypage de langages de programmation avec contraintes*. Thèse de Doctorat d'Université, Université Henri Poincaré - Nancy 1, octobre 1994.