Electronic Notes in Theoretical Computer Science 5 (1997) *to appear*

ELAN: A logical framework based on computational systems

Peter Borovanský Claude Kirchner Hélène Kirchner
Pierre-Etienne Moreau Marian Vittek

INRIA Lorraine & CRIN-CNRS
615, rue du Jardin Botanique, BP 101
54602 Villers-lès-Nancy Cedex, France
`http://www.loria.fr/equipe/protheo.html`

Abstract

ELAN implements computational systems, a concept that combines rewriting logic with the powerful description of rewriting strategies. ELAN can be used either as a logical framework or to describe and execute deterministic as well as non-deterministic rule based processes. We present the general features of the language and outline some of the applications it has been used for.

1 Introduction

> **elan** n. **1**. Enthusiastic vigor and liveness. **2**. Style; flair. [Fr < OFr. *eslan*, rush < eslancer, to throw out: es-, out (< Lat. ex-) + *lancer*, to throw (< LLat. *lanceare*, to throw a lance < Lat. *lancea*, lance).]
> The American Heritage Dictionary

Starting from the idea that inference systems can be quite conveniently described by rewrite rules, we began in the early nineties the design and implementation of a language in which inference systems can be represented in a natural way, and executed reasonably efficiently. This led us quickly to formalise such a language using the conditional rewriting logic introduced by J. Meseguer [Mes92] and to see ELAN as a logical framework where the frame logic is rewriting logic [Vit94].

In ELAN, a logic can be expressed by specifying its syntax and its inference rules. The syntax of the logic can be described using mixfix operators as in the OBJ or Maude languages [GKK$^+$87,CELM96]. The inference rules of the logic are described by conditional rewrite rules. In order to make the description executable, we introduced the notion of strategy [KKV95a]. A computational system consists of a rewriting theory plus a strategy description. The underlying concepts are thus extremely simple and a natural question was to understand how far one can go in this direction, and how usable such a framework is. From the implementation of an interpretor first and recently of a compiler, we have experimented with the system many examples from small classical ones to large and complex ones. To summarize, ELAN provides the following main features:

- A semantics based on many-sorted rewriting logic,
- A powerful language to express strategies of rewrite rule application, including dont-care and dont-know choices on strategies,
- A general pre-processor making easier the translation of a logic into rewriting logic,
- A standard library to facilitate user developments,
- Modular constructions via local or global importations as well as parametric modules,
- A generic mixfix and user-definable syntax,
- Associative commutative (AC) operators in interpreted mode,
- A very efficient compiler for ELAN programs without AC operators.

The goal of this paper is to give a general presentation of the system and of some of the realisations it has been used for, in order to convince the reader that the approach is not only realistic but also extremely useful as a logical framework allowing to conduct both computations and deductions in a combined and very efficient way. After this introduction, the paper presents the general features of the ELAN language. Then we describe the interpretor and the compiler as well as their performances. We also describe the standard library as provided with the system distribution and we give a short description of some of the applications developed in the language such as constraint solving, logic programming or theorem proving.

2 A short description of ELAN

2.1 ELAN *components*

Since we wanted ELAN to be modular, with a syntactic level well-adapted to the user needs, the language is designed in such a way that programming can be done at three different levels:

- First the design of a computational system is done by the so-called *super-user*.
- Such a logic is used by the (standard) *user* in order to write a specification.
- Finally, the *end-user* evaluates queries in the specification, following the semantics described by the logic.

A simple example, which description principle is summarised in Figure 1, is the formalisation of syntactic unification where the tasks are divided as follows:

(i) the super-user describes in a generic way the unification inferences, i.e. a logic for unification, together with a strategy for the application of the inference rules,

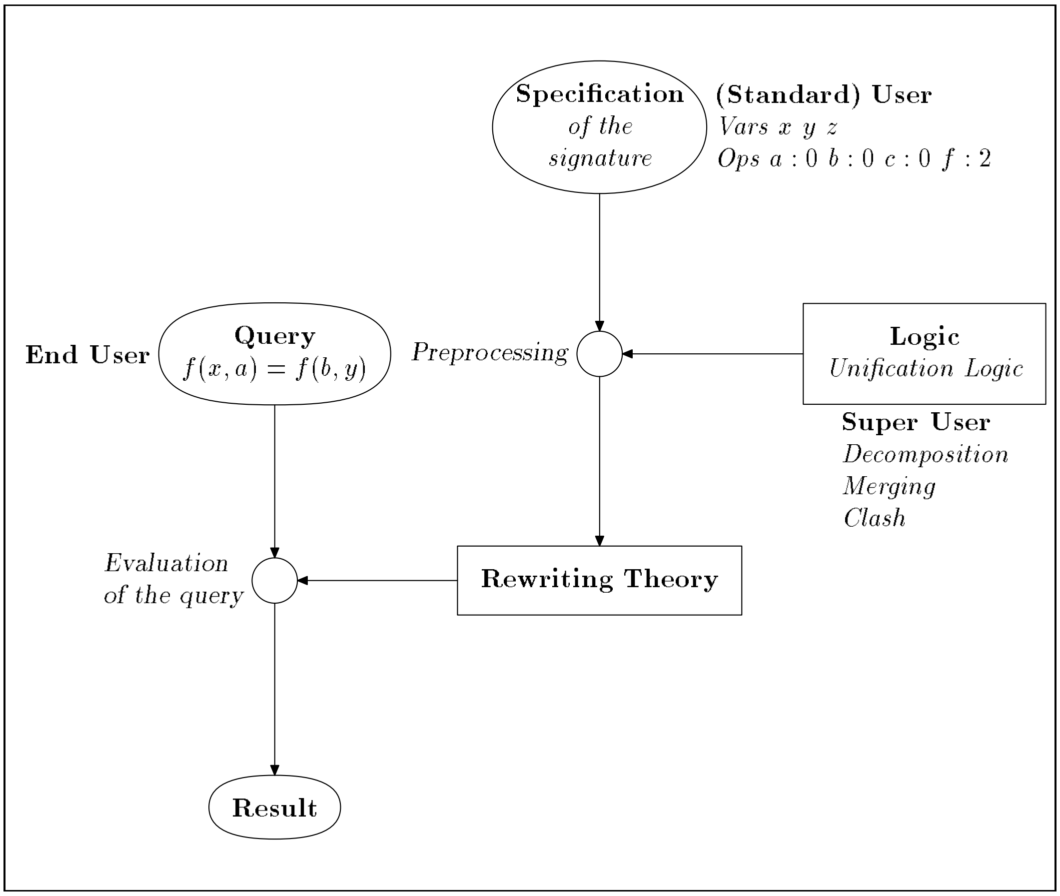

Fig. 1. The representation of syntactic unification in ELAN

(ii) the user gives the specification of an algebra in which (s)he wants to unify terms; in this case, this is quite simple since it amounts to specify the function symbols of the considered term algebra,

(iii) the end-user gives a unification problem.

The description of both the logic and the specification is done in the ELAN syntax, fully described in [KKV95b] and which can be extended by the super-user. In this place, ELAN provides a parser for any context-free grammar. This gives the ability to express grammars in a natural way, and in particular to describe mixfix syntax.

To allow a high degree of modularity, computational system descriptions for the ELAN interpreter are built from elementary pieces that are modules. A module can import other modules and defines its own signature, that is the symbols used to express the syntax of statements and queries. It defines also its own transition/rewrite rules useful to evaluate functions, and its strategies (potentially non-deterministic) for applying these rules. These descriptions are assumed to be done in files with specific extensions: `.lgi` for the top level logic description, `.eln` for a module used in a logic description, `.spc` for a specification (a program written in the defined logic).

To illustrate this, Figure 2 gives the module that the super user has to write in order to describe the derivative operation on simple polynomials. The user

```
LPL polyGeneric description

            // The syntax of the specification that could be given by the user
            // (in ELAN's terminology) is described next. In this example
            // it consists in only one part called 'Vars'
            // This fixes the syntax allowed in the file 'someVariables.spc'.
   specification description
      part Vars        of sort list[identifier]
                       modules varSyntax
   end

   query        of sort polynomial
                 result     of sort polynomial
                 modules    polyGeneric
                 start with (derive)query

end of LPL description
```

Fig. 2. A simple logic example

then gives an actual signature in which he wants to execute the logic. In this simple case described in Figure 2.1, it consists simply to define the variables on which the monomials are built. Notice that the syntax in which the user must describe the specification is fixed by the super user in the description of the current logic. The logic is using modules that contain the description of the

```
specification someVariables

   Vars              X     Y     Z

end of specification
```

Fig. 3. A simple specification example

computational system, i.e. the rewrite rules and the strategies as described for example in Figure 4.

2.2 Strategies

Strategies is one of the main originality of ELAN. Practically, a strategy is a way to describe which computations the user is interested in. A strategy specifies where a given rule should be applied in the term to be reduced. From a theoretical point of view, a strategy is a subset of all proof terms defined by the current rewrite theory. The application of a strategy to a term results in the (possibly empty) set of all terms that can be derived from the starting term using this strategy [KKV95a]. When a strategy returns an empty set of terms, we say that it *fails*.

The current version of the language allows a restricted but still powerful definition of strategies that are built in two steps. The *first level* consists of defining regular expressions built on the alphabet of rule labels. Moreover a rule can be applied using a user-defined strategy only at the top of a term. But this can be combined with a *second level* that consists of using strategies

```
module polyGeneric

            // In the next importation, the role of 'anyIdentifier' is
            // fundamental in order to be able to parse the specification
            // described in the file 'someVariables.spc'
            // and the query, without having to type quotes ("..").
import global
    Vars anyIdentifier list[identifier];

            // We now have to introduce the sort 'variable' in order to
            // make the distinction between constant and variable monomials.
sort variable monomial polynomial ;

op global

            // The next construction 'FOR EACH' is allowed by the ELAN
            // pre-processor, in order to generate automatically a rewrite theory
            // from a given logic and specification.
            // In this exemple, we introduce for each symbol given in the part
            // 'Vars' of the specification a new variable, having the same name.
      FOR EACH Id:identifier SUCH THAT Id:=(listExtract) elem(Vars) :
{ Id : variable; }
   one   : monomial;
   zero  : monomial;

            // This just says that a variable is a monomial
   @     :         ( variable ) monomial                              ;

            // This just says that a monomial is a polynomial
   @     :         ( monomial ) polynomial                            ;
   @ + @ :         ( polynomial polynomial ) polynomial;

endop

rules for polynomial
declare p1, p2, p1p, p2p : polynomial; x : variable;
bodies
   [deriveVar]   x          => one       end
   [deriveOne]   one        => zero      end
   [deriveZero]  zero       => zero      end
   [deriveSum]   p1 + p2    => p1p + p2p     where p1p := (derive)p1
                                             where p2p := (derive)p2
                                             end
end of rules

strategy derive
            dont care choose (deriveVar deriveOne deriveZero deriveSum)
end of strategy

end of module
```

Fig. 4. A simple module example

in the **where** construction of rule definitions. We will see through examples that the expressive power of strategies in ELAN is far more than just regular expressions and that, because of the second level, rules can indeed be applied everywhere in a term. Note also that the next version of ELAN will provide the general strategy definition mechanism described in [BKK96].

The application of a rewrite rule in ELAN yields, in general, several results: i.e. there are several ways to apply a given conditional rule with local assignments. This is first due to equational matching (currently only AC-matching) and second to the **where** assignment, since it may itself recursively return several possible assignments for variables, due to the use of strategies.

Thus the language provides a way to handle this non-determinism. This is done using the basic strategy operators: **dont care choose** and **dont know choose**.

For a rewrite rule $\ell : l \rightarrow r$ the strategy **dont care choose**(ℓ) returns *at most one* result which is undeterministically taken among the possible results of the application of the rule. In practice, the current implementation returns the first one.

On the contrary, if the ℓ rule is applied using the **dont know choose**(ℓ) strategy, then *all* possible results are computed and returned by the strategy. The implementation handles these several results by an appropriate back-chaining operation.

This is extended to the application of several rules: the **dont know choose** strategy results in the application of all substrategies and yields the union of all results; the application of the **dont care choose** strategy returns the set of results of the first non-failing. If all sub-strategies fail, then it fails too, i.e. it yields the empty set.

Two strategies can be concatenated: this means that the second strategy will be applied on all results of the first one. In order to allow the automatic concatenation of the same strategy, ELAN offers the two iterators **iterate** and **while**. The strategy **iterate** corresponds to applying zero, then one, then two, ... n times the strategy to the starting term, until the strategy fails. Thus (**iterate**(s))t returns $\bigcup_{\mathtt{n}=0}^{\infty}(\mathtt{s}^{\mathtt{n}})\mathtt{t}$. Notice that **iterate** returns the results one by one even when an infinite derivation exists. The strategy **while** iterates the strategy until it fails and return just the terms resulting of the last unfailing call of the strategy. It can thus be defined as (**while**(s))t $=$ $(\mathtt{s}^{\mathtt{n}})\mathtt{t}$ where $(\mathtt{s}^{\mathtt{n+1}})\mathtt{t}$ fails.

In order to illustrate how strategies work, let us consider the example consisting of the extraction of the constituents of a list:

```
    @: ( elem ) nelist
    @ . @: ( elem nelist ) nelist
    element(@): ( nelist ) elem

rules for elem
declare    e : elem;
           l : nelist;
bodies
   [extract]   element(e)        => e           end
   [extract]   element(e.l)      => e           end
   [extract]   element(e.l)      => element(l)  end
end of rules
```

It we assume furthermore that the constants a,b,c are of sort elem, then:

- (**while dont know choose**(extract) **endwhile**)(a.b.c) yields the set {a, b, c},
- (**iterate dont know choose**(extract) **enditerate**)(a.b.c) yields the set {element(a.b.c), a, element(b.c), b, element(c), c},
- (**while dont care choose**(extract) **endwhile**)(a.b.c) yields the set {a}.

2.3 The execution mechanism

The query is given by the end-user at the ELAN prompt level. There exist two kinds of rules: labelled ones like **deriveSum** in Figure 4 and unlabelled ones, like for instance:

[] *fib (n)* => *fib(n-1) + fib(n-2)* **if** *n > 1*

To evaluate a query, the ELAN interpreter repeatedly first normalises the term using unlabelled rules and then applies the transition rules according to the strategy. It works as follows:

Step 1 The current term is normalised using the unlabelled rules. This is done in order to perform functional evaluation and thus it is recommended to the user to provide a confluent and terminating unlabelled rewrite system to ensure termination and unicity of the result. This normalisation process is built-in in the evaluation mechanism and consists in a leftmost innermost normalisation. This should yield always a single result.

Step 2 Then one tries to apply on the normalised term a labelled rule following the strategy described in the logic description. This leads to a (possibly empty) set of terms. If this set is empty, then the evaluation backtracks to the last choice point; if it is not empty, then the evaluation goes on by setting a new choice point and evaluating one of the returned terms by going to **step 1**.

In a slightly more formal way, a rule

$$\ell : l \to d \quad s_1, \ldots, s_n$$

where the s_i are either **where** or **if** expressions, is applied on a term t by:

(i) Matching l against t. This computes a set of substitutions. If this set contains more than two elements, one is chosen and the other ones are stored for possible future backtracking. Let σ be the chosen substitution.

(ii) The evaluation goes on by evaluating the expressions $s_n, \ldots, s_1$, one by one and in this *order* (i.e. from n to 1).

(iii) If s_i is of the form **where** $x_i := (strat_i)t_i$, then one of the results (call it t'_i) of the application of the strategy $strat_i$ on the term t_i is chosen, and the substitution σ is extended by $x_i \mapsto t'_i$. The other results are stored for possible backtracking, and the evaluation goes on with s_{i-1}. If the strategy $strat_i$ fails on t_i, then we backtrack to the previous choice point.

(iv) If s_i is of the form **if** c_i, then the term c_i is evaluated following the normalisation strategy. If the result is the `bool` constant `true`, then one evaluates the next expression s_{i-1}, otherwise one backtracks to s_{i+1}.

This is fully described in [KKV95b] and [Vit94].

3 The ELAN environment

The ELAN language is provided in a system that encompasses an interpretor and a compiler. We give in this section an overview of their features.

3.1 The ELAN *interpretor*

The first way to use ELAN is to run the interpretor using a command like:

```
elan polyGeneric.lgi someVariables.spc
```

Then the user is prompted for a query to be reduced and the results are displayed. A top level command language allows the user to load modules, to run script files, to reduce terms, to display information on the internal state of the system, to trace at the appropriate level the execution of a strategy on a term. The interpretor offers the evaluation of associative and commutative symbols.

3.2 The ELAN *compiler*

An efficient compiler [Vit96] has been designed and implemented. One of its main originality is to allow the efficient execution of non-deterministic rewriting for any ELAN program encompassing no associative-commutative operators. Its efficiency comes mainly from a clever implementation of non-deterministic rewriting, from its memory management and from techniques like many to one matching.

To give an idea on its effectiveness, the following execution results have been obtained on a Sun Ultra Sparc 1 machine with 64MB memory under SUN OS 5.5.

Example	# rules applied	# rewrite per second	user time
Group completion	106.966	534.830	0.2
P5 completion	3.005.747	626.197	4.8
prolog queens8	26.633.873	313.339	75.5
queens8	128.949	1.289.490	0.1
queens10	3.426.635	1.223.798	2.8
primes 40000	8.983.549	6.910.422	1.3
fib33	11.405.773	6.003.038	1.9

The group completion is the rule based description of the Knuth and Bendix algorithm. P5 completion is the completion of a term rewrite system which is a variation of the standard group presentation. The example "prolog queens8" consists in the execution of the *Prolog* program queens8 under the meta description of the Prolog interpretor in ELAN. primes and (naive) fib are implementing the enumeration of bounded prime numbers and the computation of the n*th* Fibonacci number. For comparison, CamlSuperLight, the latest version of the CAML compiler runs fib 33 in 12.5 seconds.

One can notice that the number of rewrite per second varies in a wide range. This is mainly due to the facts that first, the application of non-deterministic rewrite rules is less efficient due to the associated control and that second, we have only counted the successful rewrite, and for some examples there is a huge number of unsuccessful ones due to the specific form of the rewrite system.

This shows that the rewrite concept can now be implemented in such a way that it becomes quite competitive with either functional programming (when deterministic) or logic programming (when nondeterministic), even when combining together these two concepts.

4 The standard library

In ELAN the user has the possibility to start from nothing and to create his own world, using a non-conditional rewriting logic[1]. Nevertheless in most cases, users are interested in using standard data structures to build their own ones. So we provide several standard useful built-ins described below. We also provide standard objects like terms in an ELAN written library called the "standard library". ELAN can be used without any reference to this library, *except* for what concerns the use of the built-in objects. This library has been designed to be small and as efficient as possible. In particular *no* AC operators is used. The resulting code is more efficient, at the price of sometimes heavier descriptions. But this allows using the current version of the ELAN compiler, with the advantages previously mentioned.

4.1 Built-ins

Booleans

ELAN provides the true and false values and introduces the `bool` module. These two values are built-in and are deeply connected to the implementation of conditions in rewrite rules. To enrich the booleans, *polymorphic* equality and disequality are defined and are also built-in.

Numbers

Numbers can of course be created "by hand", but we choose in ELAN to provide built-in integers and floating point computations. Floating point computations, as provided by the C compiler used for creating your ELAN version, are available using the `double` module.

Identifiers

Two important built-in modules concern identifiers. First the standard ones (i.e. without quotes) and a similar version but with quotes. In fact

[1] Indeed rewriting with conditional rules is connected to the built-in booleans since firing a rule results from a positive match and the evaluation of the condition to the built-in value true.

quoted identifiers are often used by the super user when defining a logic in order to avoid syntactic conflicts at parsing time. Unquoted identifiers are mostly used in specifications.

Elementary term computations

Since they are of primarily use in symbolic computations on terms (and remember that everything in ELAN is a term except the built-ins), several operations like taking the subterm at a given position, or replacing a subterm by another term, are provided as built-ins.

4.2 Standard ELAN *modules*

Based on the above built-ins, the following modules are provided. They are all written in ELAN and are easily modifiable.

Parameterised pairs and parameterised lists are provided with their standard strategies. Terms (with or without variables) are built as a parameterised module that uses its own reference. Note that one difficulty is that the signature is coming from the specification given by the programmer. Substitutions on terms are also provided, as well as equations, system of equations and syntactic unification.

Basic computations on atoms are available in the same spirit as for terms. Finally several modules are given for describing a possible syntax for the user specifications. More complicated syntax (e.g. mixfix) can also be defined.

5 Contributed works

This section surveys several examples that have been fully developed using ELAN. It shows that the rule-based approach of general deduction as presented in [MOM93], as well as more specific processes, like unification advocated for example in [JK91], can be realistically used in order to directly implement these concepts.

5.1 Mini Prolog and narrowing

A simple programming language based on Horn clause logic and SLD-resolution has been implemented in ELAN and is fully described in [Vit94,KKV95a].

In Horn clause logic, formulas are of the form $A \Leftarrow B_1, \ldots, B_n$ with A, B_i being atoms, and a theory is given by a signature and a set of formulas. SLD-resolution is mainly described with two rules that are direct translations of the resolution and reflection rules.

In the same vein, the constraint narrowing process has been described. Thanks to its modularity, it can be easily combined with commutative unification giving in a very simple and elegant way the first (to our knowledge) implementation of commutative narrowing.

5.2 Constraint solving

We have experimented the use of ELAN on many constraint solving mechanisms including syntactic unification (which is provided in the standard library), commutative unification but also disunification as well as certain ordering constraints based on the recursive path ordering.

More recently, ELAN has been used in formalising the consistency techniques used for the constraint satisfaction problem. As described in [Cas96], this provides a nice application of computational systems to the rule-based formalization of these techniques.

Finally let us mention the prototypal use of ELAN in the preprocessing of the finite domain constraints given to the ILOG solver [PL95]. The idea consists in transforming the input constraints into a well-suited representation to allow the best possible propagation by the solver. At this occasion, Gauss elimination, simplification and elementary computations on polynomials have been encoded in ELAN.

5.3 Constraint Solving Combination

Based on theoretical works on the modular combination of constraint solvers (see e.g. [BS95,Rin96a,KR94]), we are now using the capability of ELAN to interact with external programs in order to combine constraint solvers. The main idea developed in [Rin96b] is to incorporate built-in computations that need special data structures to be efficient. Typically, ELAN provides syntactic unification from which commutative unification can be derived just be adding another decomposition rule for commutative symbols. For AC-unification (unification modulo an associative and commutative symbol) the algorithm is quite more complicated, and needs in particular to solve linear Diophantine equations. On the other hand, quite efficient implementations of AC-unification already exist. It is thus natural to use ELAN in order to described the constraint combination logic and then to run the individual constraint solvers, either as built-ins (e.g. for AC-unification) or with their ELAN description (e.g. for commutative unification).

5.4 Completion

One of our initial goal in designing ELAN was to provide a logical framework in order to perform proof of program properties. A step toward this goal is thus to describe completion of a rewrite system in order to be able to perform proof of termination and confluence of equational specifications. This has been done in ELAN using the general approach of deduction with constraints [KKR90]. This allows having an executable description of the deduction process which is the same as the rule-based one commonly used in papers. Furthermore, the flexible strategy description allowed by ELAN gives to the implementer the possibility to experiment various completion approaches. Figure 5 gives an idea of the way we encoded it in ELAN. This is described in [KM95].

```
module constraintCompletion[vars,fss,prec,syst]

...

rules for constraintEquation
declare
        s,t,l,r,g,d              : term;
        omega                    : list[int];
        newConstraint            : constraint;
        newConstraintEq          : constraintEquation;
        c,c1                     : constraint;
        sigma                    : substitution;
bodies

    [deduce] deduce(g->d[c1], l->r[c]) => newConstraintEq
        where newConstraintEq    := (deleteVariables) g[r] at omega=d[newConstraint]
        where newConstraint      := (unifys) c & c1 & g at omega =? l
        where omega              := (nvocc) nvocc(g)
    end

...
end of rules


strategy deduce
    dont know choose(deduce)
end of strategy

end of module
```

Fig. 5. A completion rule and strategy

5.5 Higher-order unification

Considering a logical framework like ELAN based on rewriting logic which is essentially first-order, one question arises quickly: "how convenient is such a framework to express higher-order features?". A first answer is given in [DHK95], where unification in the simply typed lambda-calculus is expressed in the first-order equational calculus of explicit substitutions. This has been implemented in ELAN in a quite natural way, both in the general case [Bor95] and in the restricted situation of patterns described in [DHKP96].

6 Conclusion

The ELAN system has been designed and implemented with the general intent to understand the concrete power and usefulness of rewriting for both deduction and computation. This naturally takes place in the current stream of several works on logical frameworks. In our case, the base framework is rewriting logic. The various experiments made possible by the existing implementation have shown the fundamental interest of such a logical framework at the edge between logic and computation. The powerful notion of strategy, that we have designed and implemented, allows us to describe, in the potential search space, the deductions that one wants to explore, and to guide the computations so that they become more efficient.

This work and many others stemming from the seminal idea of rewriting logic lead the way to an exciting research field, where many general questions are arising, among which we can mention:

- the use of computational systems for the design of an integrated proof environnement where the programs, the proofs, the provers and their proof plans, but also specific decision procedures, can all be designed and executed in the same uniform framework,
- the use of rewriting logic for inductive reasoning,
- the reflective power of rewriting logic,
- the complexity of mapping from a given logic to rewriting logic,
- the relationships between linear and rewriting logics,
- the comparison with other logical frameworks [2].

The work on term rewriting, started now 25 years ago, allowed very strong theoretical interests and results to emerge. But it has encountered sceptiscism because of the relative inefficiency of rewriting as a computation and a fortiori as a deduction process. One main originality of ELAN is its capability to perform deterministic and non-deterministic computations, thanks to the dont-know and dont-care choose operators on strategies. The current implementation of ELAN with in particular its compiler, shows that the inefficiency criticism is no more valid, since deterministic as well as undeterministic rewriting can be executed as fast as the best compiled logical languages like ML or CLP. We get now the evidence that rewriting is combining the descriptive and computational power needed for many applications in a unique yet very simple concept. Thus, because of the crucial role played by rewriting in this context, many of the questions studied during the last decades surface again under the more genral point of view of transition systems, since the intended semantics is no more restricted to the (up to now) standard equational interpretation.

From the more specific point of view of ELAN, several works are under development. Let us mention some of them.

- A system for performing input/ouput is currently being designed [Vir96].
- We are investigating the extension of the language, and its interpretor and compiler, to the order-sorted rewriting case with built-in theories like associativity and commutativity.
- Since quite complicated proofs can be achieved using the system (for example the full trace of the execution of the completion process mentioned above could be as big as several 100 Mo), appropriate tools for editing and understanding such proofs are under design.
- Internalizing the proof terms and the strategies in the logic is an extremely powerful idea that allows in particular to define strategies using the concept of computational system. Investigations in this direction are described in [BKK96].
- A first indication of the reflective power of a framework is to be able to express its own evaluation. In ELAN, it is possible to express in a very

[2] See the web-page at `http://www.cs.cmu.edu/afs/cs/user/fp/www/lfs.html`

natural way the rewriting process itself [Vit94,KKV95a]. We are now investigating the reflective power of the rewriting logic in this context and the introduction of reflective capabilities in ELAN, as proposed in [KM96].

- Rewriting logic is a natural framework to express concurrency, but conversely it it quite challenging to use concurrent rewriting to efficiently execute deductions and computations. We have first investigated this line of research in [KV90], designing an implementation of fine grained concurrent rewriting together with its garbage collector [Alo95] for unconditional as well as conditional rewriting [AK96]. The implementation ReCo runs on various parallel architectures and we currently investigate the definition of specific strategies for concurrent rewriting.

Further information on the ELAN system, including the current distribution of the system, can be found at the following address:
`http://www.loria.fr/equipe/protheo.html/PROJECTS/ELAN/elan.html`.

Acknowledgment: We thank for their interaction in the design and the implementation of ELAN, Jounaidi Benhassen, Carlos Castro, Christophe Ringeissen, Patrick Viry and the Protheo team. We also thank José Meseguer for our long collaboration on many topics concerning the theoretical background as well as the implementation of ELAN.

References

[AK96] I. Alouini and C. Kirchner. Toward the concurrent implementation of computational systems. In M. Hanus, editor, *Proceedings of ALP'96*, Lecture Notes in Computer Science, Aachen (Germany), September 1996. Springer-Verlag.

[Alo95] I. Alouini. Concurrent garbage collection for concurrent rewriting. In J. Hsiang, editor, *Proceedings 6th Conference on Rewriting Techniques and Applications, Kaiserslautern (Germany)*, volume 914 of *Lecture Notes in Computer Science*, pages 132–146. Springer-Verlag, 1995.

[BKK96] P. Borovanský, C. Kirchner, and H. Kirchner. Controlling rewriting by rewriting. In J. Meseguer, editor, *Proceedings of the first international workshop on rewriting logic*, volume 5, Asilomar (California), September 1996. Electronic Notes in Theoretical Computer Science.

[Bor95] P. Borovanský. Implementation of higher-order unification based on calculus of explicit substitutions. In M. Bartošek, J. Staudek, and J. Wiedermann, editors, *Proceedings of the SOFSEM'95: Theory and Practice of Informatics*, volume 1012 of *Lecture Notes in Computer Science*, pages 363–368. Springer-Verlag, 1995.

[BS95] F. Baader and K. U. Schulz. Combination of constraint solving techniques: An algebraic point of view. In J. Hsiang, editor, *Rewriting Techniques and Applications, 6th International Conference, RTA-95*, LNCS 914, pages 352–366, Kaiserslautern, Germany, April 5–7, 1995. Springer-Verlag.

[Cas96] C. Castro. Solving Binary CSP using Computational Systems. In J. Meseguer, editor, *Proceedings of the first international workshop on rewriting logic*, volume 5, Asilomar (California), September 1996. Electronic Notes in Theoretical Computer Science.

[CELM96] M. Clavel, S. Eker, P. Lincoln, and J. Meseguer. Principles of Maude. In J. Meseguer, editor, *Proceedings of the first international workshop on rewriting logic*, volume 5, Asilomar (California), September 1996. Electronic Notes in Theoretical Computer Science.

[DHK95] G. Dowek, T. Hardin, and C. Kirchner. Higher-order unification via explicit substitutions, extended abstract. In D. Kozen, editor, *Proceedings of LICS'95*, pages 366–374, San Diego, June 1995.

[DHKP96] G. Dowek, T. Hardin, C. Kirchner, and F. Pfenning. Unification via explicit substitutions: The case of higher-order patterns. In M. Maher, editor, *Proceedings of JICSLP'96*, Bonn (Germany), September 1996. The MIT press.

[GKK+87] J. A. Goguen, C. Kirchner, H. Kirchner, A. Mégrelis, J. Meseguer, and T. Winkler. An introduction to OBJ-3. In J.-P. Jouannaud and S. Kaplan, editors, *Proceedings 1st International Workshop on Conditional Term Rewriting Systems, Orsay (France)*, volume 308 of *Lecture Notes in Computer Science*, pages 258–263. Springer-Verlag, July 1987. Also as internal report CRIN: 88-R-001.

[JK91] J.-P. Jouannaud and C. Kirchner. Solving equations in abstract algebras: a rule-based survey of unification. In J.-L. Lassez and G. Plotkin, editors, *Computational Logic. Essays in honor of Alan Robinson*, chapter 8, pages 257–321. The MIT press, Cambridge (MA, USA), 1991.

[KKR90] C. Kirchner, H. Kirchner, and M. Rusinowitch. Deduction with symbolic constraints. *Revue d'Intelligence Artificielle*, 4(3):9–52, 1990. Special issue on Automatic Deduction.

[KKV95a] 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. The MIT press, 1995.

[KKV95b] C. Kirchner, H. Kirchner, and M. Vittek. ELAN *V 1.17 User Manual*. Inria Lorraine & Crin, Nancy (France), first edition, November 1995.

[KM95] H. Kirchner and P.-E. Moreau. Prototyping completion with constraints using computational systems. In J. Hsiang, editor, *Proceedings 6th Conference on Rewriting Techniques and Applications, Kaiserslautern (Germany)*, volume 914 of *Lecture Notes in Computer Science*, pages 438–443. Springer-Verlag, 1995.

[KM96] H. Kirchner and P.-E. Moreau. Computational reflection and extension in elan. In J. Meseguer, editor, *Proceedings of the first international workshop on rewriting logic*, volume 5, Asilomar (California), September 1996. Electronic Notes in Theoretical Computer Science.

[KR94] H. Kirchner and C. Ringeissen. Combining symbolic constraint solvers on algebraic domains. *Journal of Symbolic Computation*, 18(2):113–155, 1994.

[KV90] C. Kirchner and P. Viry. Implementing parallel rewriting. In P. Deransart and J. Maluszynski, editors, *Proceedings of PLILP'90*, volume 456 of *Lecture Notes in Computer Science*, pages 1–15, Linköping (Sweden), August 1990. Springer-Verlag.

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

[MOM93] N. Martí-Oliet and J. Meseguer. Rewriting logic as a logical and semantic framework. Technical Report SRI-CSL-93-05, SRI International, Computer Science Laboratory, August 1993. To appear in D. Gabbay, ed., *Handbook of Philosophical Logic*, Oxford University Press.

[PL95] J.-F. Puget and M. Leconte. Beyond the black box,: Constraints as objects. In J. Loyd, editor, *Proceedings of the International Logic Programming Symposium*, pages 513–527. The MIT press, 1995.

[Rin96a] C. Ringeissen. Combination of matching algorithms. *Information and Computation*, 1996.

[Rin96b] C. Ringeissen. Prototyping combination of unification algorithms with ELAN. Technical report, CRIN, 1996.

[Vir96] P. Viry. Input/output for rule based languages. Technical report, Universita di Pisa, 1996.

[Vit94] 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, October 1994.

[Vit96] M. Vittek. A compiler for nondeterministic term rewriting systems. In H. Ganzinger, editor, *Proceedings of RTA'96*, volume 1103 of *Lecture Notes in Computer Science*, pages 154–168, New Brunswick (New Jersey), July 1996. Springer-Verlag.