URL: http://www.elsevier.nl/locate/entcs/volume15.html 16 pages

An Overview of ELAN [1]

Peter Borovanský, Claude Kirchner, Hélène Kirchner
Pierre-Etienne Moreau, Christophe Ringeissen

LORIA – CNRS, INRIA and UHP
Campus Scientifique BP 239
F-54506 Vandoeuvre-lès-Nancy Cedex
E-mail: elan@loria.fr

Abstract

This paper presents a comprehensive introduction to the ELAN rule-based programming language. We describe the main features of the language, the ELAN environment, and introduce bibliographic references to various papers addressing foundations, implementation and applications of ELAN.

1 Introduction

The ELAN system [18] provides an environment for specifying and prototyping deduction systems in a language based on rules controlled by strategies. Its purpose is to support the design of theorem provers, logic programming languages, constraints solvers and decision procedures and to offer a modular framework for studying their combination.

ELAN takes from functional programming the concept of abstract data types and the function evaluation principle based on rewriting. But rewriting is inherently non-deterministic since several rules can be applied at different positions in a same term, and in ELAN, a computation may have several results. This aspect is taken into account through choice operations and a backtracking capability. One of the main originality of the language is to provide strategy constructors to specify whether a function call returns several, at-least one or only one result. This declarative handling of non-determinism is part of a strategy language allowing the programmer to specify the control on rules application. This is in contrast to many existing rewriting-based languages where the term reduction strategy is hard-wired and not accessible to the designer of an application. The strategy language offers primitives for

[1] This work has been partially supported by the Esprit Basic Research Working Group 22457 - Construction of Computational Logics II.

sequential composition, iteration, deterministic and non-deterministic choices of elementary strategies that are labelled rules. From these primitives, more complex strategies can be expressed. In addition the user can introduce new strategy operators and define them by rewrite rules. Evaluation of strategy application is itself based on rewriting. So the simple and well-known paradigm of rewriting provides both the logical framework in which deduction systems can be expressed and combined, and the evaluation mechanism of the language.

The purpose of this paper is to summarize ELAN features, library and environment and to provide a guide to the literature on the language.

The main features of the ELAN language are presented in Section 2. The current version of ELAN includes an interpreter and a compiler written respectively in C++ and Java, a library of standard ELAN modules, a user manual and examples of applications. The different components of the environment are described in Section 3. Section 4 provides a commented bibliography on foundations, implementation and applications of ELAN. Section 5 shortly compares ELAN with the existing rule-based systems. More informations together with the current version of the system can be found on the WEB site [2].

2 The specification language

The specification formalism provided in the ELAN system is close to the algebraic specification formalism. Signatures introduce sorts of data and operations applied of them. One particularity of ELAN is to provide mixfix syntax for operators. For defining Booleans for instance, a sort `Bool` is declared inhabited by two constants `true` and `false`. Boolean terms are constructed with operators `and, or, not`. Attributes `assocLeft`, `assocRight` and `AC` may be used to declare an operator as left-associative, right-associative for parsing purposes, or associative and commutative. The associativity and commutativity axioms are called structural axioms and their application is embedded into the matching process. Priorities may be defined too, using the attribute `pri`, and aliased syntactic forms for an operator are introduced with an attribute `alias`.

[2] http://www.loria.fr/equipes/protheo/PROJECTS/ELAN/elan.html.

```
module boolean
sort Bool; end
operators global
  true      :             Bool;
  false     :             Bool;
  @ and @   : (Bool Bool) Bool assocLeft pri 100;
  @ or @    : (Bool Bool) Bool assocLeft pri 100;
  (@ and @) : (Bool Bool) Bool assocLeft pri 100 alias @ and @:;
  (@ or @)  : (Bool Bool) Bool assocLeft pri 100 alias @ or @:;
  not @     : (Bool )     Bool           pri 200;
  not (@)   : (Bool )     Bool           pri 200 alias not @:;
end
```

It may sometimes be useful to define an injection of one sort into another. This is done in ELAN with an anonymous operator. Assume that we want to define a sort `Constraint` embedding the sort `Bool`, so that any boolean formula becomes a constraint. This is expressed in ELAN as follows:

```
module constraint
sort Constraint; end
operators global
  @ : (Bool) Constraint;
end
```

In the algebraic style, the semantics of operations on data is described by a set of first-order formulas. In ELAN, the formulas are a very general form of rewrite rules with conditions and local evaluations. For instance, simple rewrite rules for booleans are given as follows:

```
rules for Bool
  P : Bool;
global
  [] true or P   => true      end
  [] false or P  => P         end
  [] true and P  => P         end
  [] false and P => false     end
  [] not true    => false     end
  [] not false   => true      end
end
```

The two values `true` and `false` are said irreducible or in normal form. This set of rules is terminating and confluent, which ensures that any boolean formula has a unique normal form. For such systems, it is not needed to specify in which order the rules are applied, nor at which position in the term. The ELAN system adopts in such a case a strategy by default which selects the leftmost and innermost redex at each step. However in many situations, and especially to deal with non-confluent or non-terminating rewrite systems, it is suitable to express which rule to apply. For specifying this kind of control, ELAN introduces the possibility to name rules, using brackets in front of a rule to enclose its name. In the previous boolean example, the names are unspecified

and such rules are said unlabelled. The capability of specifying control is a main originality of ELAN compared to other specification languages. Let us explain in more details how to build strategies that compute one or several results, specify the order of applied rules, or iterate as much as possible the application of a strategy or a rule on a term.

2.1 Strategies Specification

A labelled rule is the most elementary strategy and is called a primal strategy. The result of applying a rule labelled lab on a term t is a set of terms. Note that there may be several rules with the same label. If no rule labelled lab applies on the term t, the set of results is empty and we say that the rule lab fails. To understand why applying one rule at the top of a term can yield several results, one has to know that local assignments in a rewrite rule can call strategies on subterms. If the strategy in a local assignment has several results, so has the rewrite rule. A labelled rule lab can be considered as the simplest form of a strategy which returns all results of the rule application. As any strategy, lab can also be encapsulated by an operator dc one that returns a non-deterministically chosen result. In that case, dc one(lab) returns at most one result. In addition ELAN provides a few built-in strategy operators that take possibly several strategies as arguments and can be used to build new strategies:

- the concatenation operator denoted ; builds the sequential composition of two strategies S_1 and S_2. The strategy $S_1; S_2$ fails if S_1 fails, otherwise it returns all results (maybe none) of S_2 applied to the results of S_1;
- the dk operator, with a variable arity, is an abbreviation of *dont know choose*. dk$(S_1, \ldots, S_n)$ takes all strategies given as arguments, and returns, for each of them the set of all its results. dk$(S_1, \ldots, S_n)$ fails if all strategies $S_1, \ldots, S_n$ fail.
- the dc operator, with a variable arity, is an abbreviation of *dont care choose*. dc$(S_1, \ldots, S_n)$ selects only one strategy that does not fail among its arguments, say S_i, and returns all its results. dc$(S_1, \ldots, S_n)$ fails if all strategies $S_1, \ldots, S_n$ fail. How to choose S_i is not specified.
- a specific way to choose an S_i is provided by the first operator that selects the first strategy that does not fail among its arguments, and returns all its results. So if S_i is selected, this means that all strategies $S_1, \ldots, S_{i-1}$ have failed. Again first$(S_1, \ldots, S_n)$ fails if all strategies $S_1, \ldots, S_n$ fail.
- if only one result is wanted, one can use the operators first one or dc one that select a non-failing strategy among their arguments (either the first or anyone respectively), and return a non-deterministically chosen result of the selected strategy.
- id is the identity strategy that does nothing, and never fails.
- fail always fails and returns an empty set of results.

- repeat*(S) iterates the strategy S until it fails and then returns the last obtained result. repeat*(S) never fails and terminates only when S fails.
- iterate*(S) is similar to repeat*(S), except that it returns all intermediate results of successive applications of S.

In addition to these primitive strategy operators, the user can define new strategy operators and strategy rules for their evaluation [3].

2.2 Non-deterministic Computations

In order to illustrate these constructions, let us write a program that computes the images by a given function `f` of every element occurring in an input list. Lists are defined in the following module, where the empty list is denoted by `nil` and the concatenation operator by `.` :

```
module list
sort Element List; end
operators global
  a          :                 Element;
  b          :                 Element;
  c          :                 Element;
  nil        :                 List;
  @.@        : (Element List) List;
end
```

The list composed of elements `a` and `b` is thus represented by the term `a.b.nil`. Given a function `f(@): (Element) Element`, we define `map-f(@): (List) Element` by the following rewrite rules where `element` and `list` are respectively local variables of sort `Element` and `List`.

```
  [head]  map-f(element.list) => f(element)       end
  [tail]  map-f(element.list) => map-f(list)      end
```

Then we introduce a constant strategy operator `mapStrat` defined by the following strategy rule

```
[] mapStrat => iterate*(dc one(tail));dc one(head) end
```

`mapStrat` applied on `map-f(a.b.c.nil)` returns successively `f(a)`, then `f(b)` and finally `f(c)`. If `f` is defined as the identity on `Element`, we get a strategy called `listExtract` that extracts all elements of a list. This illustrates how the notion of strategy can be used to compute a set of normal forms without using explicitly a notion of set.

2.3 Rules Specification

Until now, we have seen how to apply labelled rules and strategies at the top of a term. In order to apply strategies on subterms, the syntax of rewrite rules has been enriched by local evaluations, used to call strategies, to factorise sequences of computations and to specify conditions of application. The general syntax of an ELAN rule is as follows:

<*rule*> ::= "[" [<*label*>] "]" <*term*> "=>" <*term*> { <*local evaluation*> }*

```
<local evaluation>  ::= if <boolean term>
                    |   where <variable> ":=" "(" [ <strategy> ] ")" <term>
                    |   where <term> ":=" "(" [ <strategy> ] ")" <term>
                    |   choose
                            { try { <local evaluation> }+ }+
                        end
```

After the selection of a rule by matching its left-hand side to the term to reduce, the local evaluations are evaluated in order and potentially enrich the matching substitution. If no evaluation fails, the rule can apply and the resulting term is built from the right-hand side and the enriched matching substitution. Let us see the three kinds of local evaluations which significantly increase the expressivity of rewrite rules:

- A condition is a boolean expression `c` introduced by the keyword `if`. The term `c` is put in normal form and compared to the predefined boolean value `true`. If they are equal, the condition is satisfied, and the next local evaluation is considered. Otherwise, one backtracks on the previous local evaluation.
- A local assignment `where v:=(S) t` allows calling a strategy. First the term `t` is normalised w.r.t. all unlabelled rules, then the strategy `S` is applied on its normal form. In practice only one result of `S` on `t` is computed and assigned to the variable `v`. If `S` fails, the local assignment fails too and backtracking is applied. Another result of the strategy `S` on `t` may be required by the backtracking mechanism.

  The notion of local assignment has been extended to a matching condition `where p:=(S) t` where `p` is now a term. In that case, the term `p` is matched to the result of `S` on `t`, which provides values for the variables of `p`.
- The third kind of local evaluation allows factorisation of computations, with the construction `choose try ... end`. This is especially useful when there are several rules with the same left-hand side:

```
rules for int
  x,y,z : ...
global
  [] f(x) => r1(z)
       where y:=() g(x)
       where z:=(s1) x
  end
  [] f(x) => r2(z)
       where y:=() g(x)
       where z:=(s2) x
  end
end
```

In order to reduce the term `f(a)` for instance, a rule is selected, say the first one. Assume that the strategy `s1` on the term `a` fails. In this case, the second rule is tried and the normal form of `g(a)` is re-computed. To avoid this kind of redundancy, one can use the `choose try ... end` construc-

tion and write the equivalent ELAN program which avoids this drawback:

```
rules for int
  x,y,z,result : ...
global
  [] f(x) => result
       where y:=() g(x)
       choose try where z:=(s1) x
                   where result:=() r1(z)
              try where z:=(s2) x
                   where result:=() r2(z)
       end
  end
end
```

2.4 Modularity and parameterization

ELAN is a modular language that allows parameterized modules. It provides also a more subtle parameterization through the use of preprocessing as described in the next section. Each module defines a computational system composed of sets of sorts, operators, rewrite rules, strategy operators and strategy rules. It can import other modules, via a keyword `import` followed by one or several module names. In a first approach an importation can be seen as a textual copy of the imported module in the importing one. But it is useful to specify that some operations are local and only visible in the module they belong to. The keyword `local` is used to declare that an operator or a rule is locally visible but hidden outside of the module. The dual keyword `global` makes an operator or a rule visible outside the module where it is defined. When a module is imported, the importation is itself specified as `local` or `global`. Entities visible as global in a module imported as global remain visible as global, whilst global entities become local if the module is imported as local.

3 The prototyping environment

The ELAN prototyping environment is made of several components. A library provides the user with a collection of modules that may be imported and reused in various applications. The preprocessor expands a few concise constructions allowed in the language. The parser checks the syntax of programs and verify that terms are syntactically well-formed. The interpreter is an interactive tool allowing the user to check that the results he expects are indeed obtained. The compiler transforms specifications into independent executable C code.

3.1 Library

Elementary data types such as booleans, integers, identifiers, strings are built-in. The equality and disequality operations for a given sort are built-in, as well as a few basic operations on terms (occurrence test and replacement). Recently, the standard input/output primitives have been also fully integrated in the system as built-in. A built-in module can be imported and used as any ELAN module but it only contains a declaration of built-in operations which are mapped to internal functions thanks to the special `code` attribute.

In addition, several modules implementing in ELAN useful data structures are provided in the ELAN library, such as modules defining parameterized lists, tuples and arrays. Other structures are more specific to the application domains, and the ELAN library provides modules to manipulate terms, substitutions, equational systems and to perform syntactic unification. This enumeration is not exhaustive and the library is continuously enriched by new modules written in ELAN.

A third level in the library is provided by functionalities related to the strategy language. The syntactic constructions of the strategy language which do not depend on the user specifications are described in an ELAN module of the library. Operations to dynamically create typed strategies are provided [3,1]. This module dedicated to user-defined strategies has to be imported for using the full expressivity of the strategy language in ELAN programs.

3.2 Parser

Since ELAN allows the user to define his own syntax by giving a signature with a mixfix syntax, the syntactic analysis is already complex and it is not possible to use generators like *Lex* and *Yacc*, except for the fixed part of the syntax which excludes user-defined terms. So the Earley algorithm is used to analyse the part of programs which depends on the user-defined syntax for terms, while the other part is analysed by an automaton generated by a Yacc-like tool.

3.3 Preprocessor

The ELAN syntax provides a few fancy constructions: for instance the construction `P {& s_I=t_I}_I=1...3` is processed into `P & s_1=t_1 & s_2=t_2 & s_3=t_3` and `t_1,...,t_3` is processed into `t_1,t_2,t_3`, thanks the ELAN preprocessor that performs textual replacements. The construction

```
FOR EACH v SUCH THAT v:=(S) t : { e }
```

can be seen as a program generation feature. This construction replaces in the sequence of lexems `e` all occurrences of the variable `v` by each result of the strategy `S` applied to `t`. From the following specification written in ELAN:

```
operators global
  FOR  EACH L:list[identifier] AND F:identifier
  SUCH THAT L:=() a.b.nil      AND F:=(listExtract) extract(L) :
      { F : term; }
end
```

where `listExtract` is a variant, for the sort `list[identifier]`, of the strategy `mapStrat` previously defined. The preprocessor extracts the elements `a` and `b` from the list `a.b.nil` and creates the following declarations:

```
operators global
  a : term;  b : term;
end
```

So the preprocessor may be used to automatically generate parts of specifications used to analyse the rest of a program. It should be emphasised that it needs all the power of the ELAN interpreter to perform its task: in the previous example, the strategy `listExtract` has to be executed before further analysis. This illustrates the strong interaction between the parser, the preprocessor and the interpreter.

3.4 Interpreter

The interpreter takes a well-formed program and a well-formed query (both checked by the parser) and applies the rules and strategies defined in the program to the query. In order to find which rules can apply, the selection is guided by the top symbol of the rules: only those rules whose left-hand side has the same top symbol as the term to be reduced are selected. They are then tried in the order given in the program. Another kind of choices arbitrarily made by the interpreter is made for the strategy $\mathsf{dc}(S_1, ..., S_n)$ that should select randomly a non-failing strategy among $S_1, ..., S_n$. In practice, the interpreter selects the first one, so implements dc and first in the same way. Note however that there exists a version of ELAN which concurrently executes the n strategies and selects the first one which terminates without failure [4].

Once a set of rules is selected, a many-to-one matching algorithm is applied. When associative and commutative (AC for short) operators are involved, an external one-to-one AC-matching algorithm described in [Eke95] is called. This algorithm is not fully integrated in the interpreter, so data structure conversions are required and lower the efficiency of the AC-matching, already quite complex. Once a match is found, local evaluations are performed and if all succeed, the result term is built, taking advantage of the right-hand side of the rule and of term sharing.

[Eke95] S. Eker. Associative-commutative matching via bipartite graph matching. *Computer Journal*, 38(5):381–399, 1995.

3.5 Compiler

A first ELAN compiler was designed and presented in [29]. Experimentations made clear that a higher-level of programming is achieved when some functions may be declared as AC. However rewriting in such theories is computationally difficult and providing an efficient compiler for the language becomes a real challenge. The compilation techniques used in the design of the new ELAN compiler are the following:

- Many-to-one matching is implemented using deterministic automata.
- AC symbols are handled. In order to get a good efficiency of the compiled programs, the effort concentrated on most used patterns with at most two layers of AC symbols. Other patterns are transformed during a preprocessing step which introduces new `where` local assignments, and that also linearises the left-hand sides of rules. A Compact Bipartite Graph data structure is used to design an efficient many-to-one AC matching algorithm described in [23].
- Due to non-deterministic strategies, particular choice point management is needed. For implementation of backtracking, two functions are usually required: the first one, to create a choice point and save the execution environment; the second one, to backtrack to the last created choice point and restore the saved environment. Two flow control functions, set-choice-point and fail, have been implemented in assembly language. set-choice-point sets a choice point, and the computation goes on. The fail function performs a jump into the last call of set-choice-point. Their implementation is described in [24].
- More efficiency is also achieved thanks to determinism analysis. The determinism analysis phase of the ELAN compiler annotates every rule and strategy in the program with its determinism mode for use in later phases of the compiler: matching phase, various optimisations on the generated code and detection of non termination. This work is developed in [22].

3.6 An Exchange Format

An exchange format for ELAN programs, called REF format for "Reduced ELAN Format" [5], provides a representation of programs that can be shared by the different tools of the ELAN environment such as the parser, the interpreter, the preprocessor and the compiler. Such a format facilitates the connection to other systems, like ASF+SDF, to execute ELAN programs with ASF+SDF or conversely. Another advantage of REF format is to provide a term-like representation that can be easily handled by ELAN programs (a module specifying the REF format is available in the library). For example, the unlabelled rules of an ELAN program may be completed using a completion process to provide a confluent and terminating rewrite system. Such a representation and functionalities that allow to access and modify different

parts of the format, are the basis for implementing reflection in ELAN.

4 More Insights

4.1 Foundations

The logical foundations of ELAN are described in several papers. The initial ideas are presented in [18] where the notion of computational systems is introduced in a constraint solving context. The first design of ELAN is described in M. Vittek's PhD thesis [28] and in [9]. In this initial approach, strategies are expressed using the strategy constructors dc, dk, concatenation and iteration. The idea of a user-defined strategy language implemented in rewriting logic for ELAN is presented in [6,7,11]. A functional view of rewriting and strategies is given in [8] and provides a functional semantics for ELAN. Ideas on meta-interpretation and partial evaluation of the strategy language are explained in [11]. Preliminary ideas to build a reflective extension of ELAN have been presented in [21].

4.2 Implementation

The ELAN manual for the version 3.0 is available as [10]. Following [29] that describes the implementation of the first ELAN compiler, compilation techniques for Associative-Commutative normalisation are studied in [25], while more details on compilation of matching are given in [23], and non-determinism management in [22]. The REF format and its use for implementation are described in [5].

4.3 Applications

Many computational processes in automated deduction can be expressed as instances of a general schema that consists of applying transformation rules on formulas with some strategy, until reaching specific normal forms. Such processes are naturally modelised in ELAN. Several applications have been designed and are classified according to the area of interest, namely programming, proving and solving.

Programming: One of the first application was to prototype the fundamental mechanisms of logic and functional programming languages like first-order resolution and λ-calculus. The general framework of Constraint Logic Programming [JM94] can be easily designed in the ELAN framework [19], since its operational semantics is clearly formalised as rewrite rules, although the application strategy is often defined in an informal way. Some implementations [2] related to a calculus of explicit substitutions (the first-order rewrite system $\lambda\sigma$ that mimics λ-calculus) open the way of implementing higher-order logic

[JM94] J. Jaffar and M. Maher. Constraint Logic Programming: a Survey. *Journal of Logic Programming*, 19,20:503–581, 1994.

programming languages via a first-order setting. Another calculus of explicit substitutions based on the π-calculus is used to provide a formal specification of Input/Output for ELAN [27].

Proving: ELAN was used in order to implement a predicate prover based on the rules proposed by J.-R. Abrial, and implemented in the B-tools [Abr96]. We developed also a propositional sequent calculus, completion procedures for rewrite systems [20], sufficient conditions for the termination problem [17]. A library for automata construction and manipulation has been designed. Approximation automata are used to check conditions for reachability, sufficient completeness, absence of conflicts in systems described by non-conditional rewrite rules [16].

Solving: The notion of rewriting controlled by strategies is used in [19] to describe in a unified way the constraint solving mechanism as well as the meta-language needed to manipulate the constraints. This provides programs that are very close to the proof theoretical setting used now to describe constraint manipulations like unification or numerical constraint solving. ELAN offers a constraint programming environment where the formal description of a constraint solver is directly executable. ELAN has been tested on several examples of constraint solvers for various computation domains and combinations like abstract domains [19,26] (term algebras) and more concrete ones (booleans, integers, reals). In [12–15], it is shown how to use computational systems as a general framework for handling Constraint Satisfaction Problems (CSP for short). The approach leads to the design in ELAN of COLETTE, a solver for constraints over integers and finite domains.

5 Related Systems

ELAN has similarities with other rule-based systems like ASF+SDF [DHK96], CafeObj [FN96], and Maude [CDE+98]. Compared to them, ELAN has some positive aspects:

• ELAN was the first programming language based on rewriting logic integrating a notion of strategy relevant to express non-deterministic computations. More recently, analogous notions have been proposed for Maude and ASF+SDF.

• A fast ELAN compiler is available since a couple of years and the actual ver-

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

[DHK96] A. Deursen, J. Heering, and P. Klint. *Language Prototyping*. World Scientific, 1996. ISBN 981-02-2732-9.

[FN96] K. Futatsugi and A. Nakagawa. An Overview of Cafe Project. In *Proceedings of First CafeOBJ Workshop*, Yokohama (Japan), August 1996.

[CDE+98] M. Clavel, F. Durn, S. Eker, P. Lincoln, and J. Meseguer. An Introduction to Maude (Beta Version). Technical report, SRI International, Computer Science Laboratory, Menlo Park, (CA, USA), March 1998.

sion, written in Java, encompasses an improved garbage collector and handles specifications involving AC-symbols. Due to the promising results obtained with compilation techniques for rewriting, the development of compilers for rule-based languages has attracted considerable interest in the ASF+SDF and CafeObj communities. A common exchange format, like REF, might help in the development of a language-independent compiler for rewrite specifications.

• The ELAN system provides a powerful parser and original pre-processing facilities that allow writing modular and parametric specifications in a user-friendly syntax. The preprocessor is very useful for writing specifications in a concise and generic way.

On the other hand, ELAN has also a few weaknesses with respect to these other systems. First of all, the use of an exchange format for ELAN is quite new, and the REF format should only be considered as a first attempt. On this point, there is much more expertise in ASF+SDF with the AsFix format and the toolBus architecture [vdBHK97] for the interconnection of AsFix-based tools. Since the ASF+SDF system has been designed as a meta-environment for prototyping programming languages, a formalism for the syntax definition has been carefully worked out. The ELAN system is comparable to the ASF part, but does not incorporate the facilities to specify user-defined lexical entities.

An exchange format is also a possible solution to deal with reflection facilities which are already fully integrated into Maude [Cla98]. Moreover, Maude allows several possible equational axioms on user-defined function symbols like associativity, commutativity, identity, idempotency,... and their combinations, whilst ELAN only handles associativity-commutativity (possibly a combination of different AC-symbols).

Behavioural specifications with hidden sorts and states, order-sorted sorts, and object orientation, provided in CafeObj, have not been integrated in ELAN. All these points provide potential further improvements of ELAN.

ELAN References

[1] P. Borovanský. The Control of Rewriting: Study and Implementation of a Strategy Formalism. This volume.

[2] P. Borovanský. Implementation of higher-order unification based on calculus of explicit substitutions. In *Proceedings of SOFSEM'95*, Lecture Notes in Computer Science, pages 363–368. Springer-Verlag, 1995.

[vdBHK97] M. van den Brand, J. Heering, and P. Klint. Renovation of the Old ASF Meta-Environment – Current State of Affairs. In *Proceedings of International Workshop on Theory and Practice of Algebraic Specifications ASF+SDF 97, Amsterdam (The Nederlands)*, Workshops in Computing. Springer-Verlag, 1997.

[Cla98] M. Clavel. *Reflection in general logics, rewriting logic, and Maude.* PhD thesis, University of Navarre, Spain, 1998.

[3] P. Borovanský. *Le contrôle de la réécriture : étude et implantation d'un formalisme de stratégies*. Thèse de Doctorat d'Université, Université Henri Poincaré - Nancy 1, octobre 1998.

[4] P. Borovanský and C. Castro. Cooperation of Constraint Solvers: Using the New Process Control Facilities of ELAN. This volume.

[5] P. Borovanský, S. Jamoussi, P.-E. Moreau, and C. Ringeissen. Handling ELAN Rewrite Programs via an Exchange Format. This volume.

[6] P. Borovanský, C. Kirchner, and H. Kirchner. Controlling Rewriting by Rewriting. In J. Meseguer, editor, *Proc. First Intl. Workshop on Rewriting Logic and its Applications*, volume 4 of *Electronic Notes in Theoretical Computer Science*, Asilomar (California), September 1996. Elsevier.

[7] P. Borovanský, C. Kirchner, and H. Kirchner. Rewriting as a Unified Specification Tool for Logic and Control:The ELAN Language. In *Proceedings of International Workshop on Theory and Practice of Algebraic Specifications ASF+SDF 97*, Workshops in Computing, Amsterdam, September 1997. Springer-Verlag.

[8] P. Borovanský, C. Kirchner, and H. Kirchner. A functional view of rewriting and strategies for a semantics of ELAN. In M. Sato and Y. Toyama, editors, *The Third Fuji International Symposium on Functional and Logic Programming*, pages 143–167, Kyoto, April 1998. World Scientific.

[9] P. Borovanský, C. Kirchner, H. Kirchner, P.-E. Moreau, and M. Vittek. ELAN: A logical framework based on computational systems. In J. Meseguer, editor, *Proc. First Intl. Workshop on Rewriting Logic and its Applications*, volume 4 of *Electronic Notes in Theoretical Computer Science*, Asilomar (California), September 1996. Elsevier.

[10] P. Borovanský, C. Kirchner, H. Kirchner, P.-E. Moreau, and M. Vittek. ELAN *V 3.0 User Manual*. Inria Lorraine & Loria, Nancy (France), second edition, January 1998.

[11] P. Borovanský and H. Kirchner. Strategies of ELAN: meta-interpretation and partial evaluation. In *Proceedings of International Workshop on Theory and Practice of Algebraic Specifications ASF+SDF 97*, Workshops in Computing, Amsterdam, September 1997. Springer-Verlag.

[12] C. Castro. Binary CSP Solving as an Inference Process. In *Proceedings of the Eighth International Conference on Tools in Artificial Intelligence, ICTAI'96, Toulouse, France*, pages 462–463, November 1996.

[13] C. Castro. Solving Binary CSP using Computational Systems. In J. Meseguer, editor, *Proc. First Intl. Workshop on Rewriting Logic and its Applications*, volume 4 of *Electronic Notes in Theoretical Computer Science*, Asilomar (California), September 1996. Elsevier.

[14] C. Castro. Constraint Manipulation using Rewrite Rules and Strategies. In A. Drewery, G.-J. M. Kruijff, and R. Zuber, editors, Proceedings of the Second

ESSLLI Student Session, 9th European Summer School in Logic, Language and Information, ESSLLI'97, pages 45–56, Aix-en-Provence, France, August 1997.

[15] C. Castro. Building Constraint Satisfaction Problem Solvers Using Rewrite Rules and Strategies. *Fundamenta Informaticae*, 34(3):263–293, September 1998.

[16] T. Genet. Decidable Approximations of Sets of Descendants and Sets of Normal Forms. In *Proceedings 9th Conference on Rewriting Techniques and Applications, Tsukuba (Japan)*, volume 1379 of *Lecture Notes in Computer Science*, pages 151–165. Springer-Verlag, 1998.

[17] T. Genet and I. Gnaedig. Termination Proofs using gpo Ordering Constraints. In M. Bidoit and M. Dauchet, editors, *Proceedings 22nd International Colloquium on Trees in Algebra and Programming, Lille (France)*, volume 1214 of *Lecture Notes in Computer Science*, pages 249–260. Springer-Verlag, 1997.

[18] 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.

[19] C. Kirchner and C. Ringeissen. Rule-Based Constraint Programming. *Fundamenta Informaticae*, 34(3):225–262, September 1998.

[20] 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.

[21] H. Kirchner and P.-E. Moreau. A reflective extension of ELAN. In J. Meseguer, editor, *Proc. First Intl. Workshop on Rewriting Logic and its Applications*, volume 4 of *Electronic Notes in Theoretical Computer Science*, Asilomar (California), September 1996. Elsevier.

[22] H. Kirchner and P.-E. Moreau. Non-deterministic Computations in ELAN, 1998. Submitted.

[23] P. Moreau and H. Kirchner. A compiler for rewrite programs in associative-commutative theories. In C. Palamidessi, H. Glaser, and K. Meinke, editors, *Principles of Declarative Programming*, number 1490 in Lecture Notes in Computer Science, pages 230–249. Springer-Verlag, September 1998.

[24] P.-E. Moreau. A choice-point library for backtrack programming. JICSLP'98 Post-Conference Workshop on Implementation Technologies for Programming Languages based on Logic, June 1998.

[25] P.-E. Moreau and H. Kirchner. Compilation Techniques for Associative-Commutative Normalisation. In *Proceedings of International Workshop on Theory and Practice of Algebraic Specifications ASF+SDF 97*, Workshops in Computing, Amsterdam, September 1997. Springer-Verlag.

[26] C. Ringeissen. Prototyping Combination of Unification Algorithms with the ELAN Rule-Based Programming Language. In *Proceedings 8th Conference on Rewriting Techniques and Applications, Sitges (Spain)*, volume 1232 of *Lecture Notes in Computer Science*, pages 323–326. Springer-Verlag, 1997.

[27] P. Viry. Input/Output for ELAN. In J. Meseguer, editor, *Proc. First Intl. Workshop on Rewriting Logic and its Applications*, volume 4 of *Electronic Notes in Theoretical Computer Science*, Asilomar (California), September 1996. Elsevier.

[28] 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.

[29] M. Vittek. A Compiler for Nondeterministic Term Rewriting Systems. In H. Ganzinger, editor, *Proceedings 7th Conference on Rewriting Techniques and Applications, New Brunswick (New Jersey, USA)*, volume 1103 of *Lecture Notes in Computer Science*, pages 154–168. Springer-Verlag, July 1996.