Modelling Planning Problems with Rules & Strategies

Hubert Dubois, Hélène Kirchner

Modelling Planning Problems with Rules and Strategies

Hubert Dubois — Hélène Kirchner
LORIA-CNRS,INRIA,UHP
Campus Scientifique BP 239
54506 Vandoeuvre-les-Nancy Cedex
Email : {hdubois,hkirchne}@loria.fr
Web : www.loria.fr/{~hdubois,~hkirchne}

ABSTRACT. *The* ELAN *system provides an environment for specifying and prototyping deduction systems in a language based on rewrite rules controlled by strategies. We design in* ELAN *a specific planning problem, namely a controller for printing tasks, by combining rules, strategies and constraint solving on finite domains.*

KEY WORDS : *Rewrite rules, control, strategy, planning problems.*

1. Introduction

The ELAN system [KIR 95a], provides an environment for specifying and prototyping deduction systems in a language based on rewrite rules controlled by strategies. It offers a natural and simple logical framework for the combination of the computation and deduction paradigms. It supports the design of theorem provers, logic programming languages, constraint solvers and decision procedures and it offers 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. In ELAN, a rewrite rule may be labelled, may have boolean conditions introduced by the keyword **if**, and matching conditions introduced by the keyword **where**. The evaluation mechanism also involves backtracking since in ELAN, a computation may have several results. One of the main originality is to provide 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 sequential composition, iteration, deterministic and non-deterministic choices of elementary strategies that are labelled rules. From these primitives, more complex strategies can be expressed. The user can introduce new strategy operators and define them by rewrite rules. Evaluation of strategy application is itself based on rewriting. Moreover it should be emphasised that ELAN has logical foundations based on rewriting logic [MES 92] and detailed in [BOR 96, BOR 98]. 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 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. Among those, let us mention for instance the design of rules and strategies for constraint satisfaction problems [CAS 98b], theorem proving tools in first-order logic with equality [KIR 95b, CIR 97], the combination of unification algorithms and of decision procedures in various equational theories [RIN 97, KIR 98]. More information on the system can be found on the WEB site [1].

In this paper, we address another class of problems, namely planning and scheduling problems. Our goal is to use ELAN as a decision support tool, which can simulate plan executions and explore consequences of decision-making during a planification. In Section 2, we first present ELAN and the concepts of rules and strategies offered by the system. In Section 3, we describe a specific management problem, namely a controller for printing tasks, inspired from [AND 98]. We show how the problem is formalised in ELAN by combining rules, strategies and constraint solving on finite domains. The production of plans is illustrated on an example.

2. Rules and strategies in ELAN

We assume the reader familiar with basic definitions of term rewriting given in particular in [DER 90, BAA 98]. We briefly recall and introduce notations for a few concepts that will be used along this paper.

$\mathcal{T}(\mathcal{F}, \mathcal{X})$ is the set of *terms* built from a given finite set $\mathcal{F}$ of function symbols and a denumerable set $\mathcal{X}$ of variables. Positions in a term are represented as sequences of integers. The empty sequence ϵ denotes the position associated to the top-symbol. The subterm of t at position ν is denoted $t_{|\nu}$. The replacement at position ν of the subterm $t_{|\nu}$ by t' is written $t[\nu \leftarrow t']$. The set of variables occurring in a term t is denoted by $\mathcal{V}ar(t)$. If $\mathcal{V}ar(t)$ is empty, t is called a *ground term* and $\mathcal{T}(\mathcal{F})$ is the set of ground terms. A *substitution* is an assignment from a finite subset of $\mathcal{X}$ to $\mathcal{T}(\mathcal{F}, \mathcal{X})$, written $\sigma = \{y_1 \mapsto t_1, \ldots, y_k \mapsto t_k\}$. It uniquely extends to an endomorphism on $\mathcal{T}(\mathcal{F}, \mathcal{X})$. Application of σ to t is written $t\sigma$.

2.1. *Rules*

ELAN rules provide first a rich notion of condition, called *matching condition*, also used for instance in ASF+SDF [KLI 93].

A labelled rewrite rule with matching condition denoted $[\mathsf{lab}]\ l \to r\ \mathbf{where}\ p := c$ is such that $l, r, p, c \in \mathcal{T}(\mathcal{F}, \mathcal{X})$, $\mathcal{V}ar(p) \cap \mathcal{V}ar(l) = \emptyset$, $\mathcal{V}ar(r) \subseteq \mathcal{V}ar(l) \cup \mathcal{V}ar(p)$ and $\mathcal{V}ar(c) \subseteq \mathcal{V}ar(l)$. lab is called the label.
When the term p is reduced to the boolean constant *true*, the condition is written $\mathbf{if}\ c$.

1. http://www.loria.fr/ELAN.

When the term p is reduced to a variable x, the condition is written $\mathbf{where}\ x := c$.

This notion can be generalised with a sequence of matching conditions, as in $[\mathsf{lab}]\ l \to r\ \mathbf{where}\ p_1 := c_1 \ldots\ \mathbf{where}\ p_n := c_n$ in which:

— $l, r, p_1, \ldots, p_n, c_1, \ldots, c_n \in \mathcal{T}(\mathcal{F}, \mathcal{X})$,

— $\mathcal{V}ar(p_i) \cap (\mathcal{V}ar(l) \cup \mathcal{V}ar(p_1) \cup \cdots \cup \mathcal{V}ar(p_{i-1})) = \emptyset$,

— $\mathcal{V}ar(r) \subseteq \mathcal{V}ar(l) \cup \mathcal{V}ar(p_1) \cup \cdots \cup \mathcal{V}ar(p_n)$ and

— $\mathcal{V}ar(c_i) \subseteq \mathcal{V}ar(l) \cup \mathcal{V}ar(p_1) \cup \cdots \cup \mathcal{V}ar(p_{i-1})$.

A set of rewrite rules is called a rewrite system.

To apply a syntactic rule $l \to r$ on a term t at some position ν, one looks for a matching, i.e. a substitution σ satisfying $l\sigma = t_{|\nu}$. Note that t is always considered as a ground term. The algorithm which provides the unique substitution σ, whenever it exists, is called *syntactic matching*. Once a substitution σ is found, the application of the rewrite rule consists of building the *reduced term* $t' = t[\nu \leftarrow r\sigma]$. Computing the normal form of a term t w.r.t. a rewrite system R consists of successively applying the rewrite rules of R, at any positions, until no more applies. The existence and unicity of normal forms require the rewrite system R to be terminating and confluent.

To apply a conditional rule $l \to r\ \mathbf{where}\ p := c$ on a term t, the satisfiability of the condition $\mathbf{where}\ p := c$ has to be checked before building the reduced term. Let σ be the matching substitution from l to $t_{|\nu}$. Checking the matching condition $\mathbf{where}\ p := c$ consists first of using the rewrite system R to compute the normal form c' of $c\sigma$, and then verifying that p matches the ground term c'. If there exists a matching μ, such that $p\mu = c'$, the composed substitution $\sigma\mu$ is used to build the reduced term $t' = t[\nu \leftarrow r\sigma\mu]$. Otherwise the application of the conditional rule fails. For usual boolean conditions of the form $\mathbf{if}\ c$, μ is the identity when the normal form of c is the truth value *true*.

When the rule is of the form $l \to r\ \mathbf{where}\ p_1 := c_1 \ldots\ \mathbf{where}\ p_n := c_n$, the matching substitution is successively composed with each matching μ_i from p_i to the normal forms of $c_i\sigma\mu_1 \ldots \mu_{i-1}$, for $i = 1, \ldots, n$. If one of these μ_i does not exist, the application of the rule fails.

When the left-hand side l or a pattern p of the rule contains associative commutative (AC) function symbols, AC-matching is invoked. The term l is said to AC- match another term t if there exists a substitution σ such that $l\sigma =_{AC} t$, where $=_{AC}$ is the congruence generated by associativity and commutativity axioms. In general, AC-matching can return several solutions, so the application of a rule on a term may return several reduced terms. Thus rewriting in ELAN is non-deterministic and sets of results are handled through a backtracking mechanism.

2.2. *Strategies*

In order to take into account non-determinism and sets of results, and to control rule application, the concept of strategy is introduced: a strategy is a function which, when applied to an initial term, returns a set of possible results. The strategy fails if the set is empty. To define strategies, the following strategy constructors are provi-

ded:

— A labelled rule is a primal strategy. The result of applying a rule labelled lab on a term t returns a set of terms. This primal strategy fails if the set of resulting terms is empty.

— Two strategies can be concatenated by the symbol ";", i.e. the second strategy is applied on all results of the first one. $S_1; S_2$ denotes the sequential composition of the two strategies. It fails if either S_1 fails or S_2 fails. Its results are all results of S_1 on which S_2 is applied and gives some results.

— first$(S_1, \ldots, S_n)$ chooses the first strategy S_i in the list that does not fail, and returns all its results. This strategy may return more than one result, or fails if all sub-strategies S_i fail.

— first one$(S_1, \ldots, S_n)$ selects the first result of the first strategy S_i in the list that does not fail. This strategy returns at most one result and fails if all sub-strategies fail.

— dk$(S_1, \ldots, S_n)$ chooses all strategies given in the list of arguments and for each of them returns all its results. This set of results may be empty, in which case the strategy fails.

— The strategy id is the identity that does nothing but never fails.

— fail is the strategy that always fails and never gives any result.

— repeat*(S) applies repeatedly the strategy S until it fails and returns the results of the last unfailing application. This strategy can never fail (zero application of S is possible) and may return more than one result.

— The strategy iterate*(S) is similar to repeat*(S) but returns all intermediate results of repeated applications.

Elementary strategies are defined by rules of the form `[] c => strat`, where `c` is a constant strategy operator and `strat` a term built on elementary strategy constructors. Such rules are always unlabelled. Application of such a strategy `c` on a term `t`, denoted `(c)t` is performed by a C function generated by ELAN.

Defined strategies are like ordinary terms except that they have a functional sort of the form `<s -> s'>`. Their definition is given by a strategy operator declaration and a set of rules on strategies. $[S]t$ denotes the application of the defined strategy S on the term t. The application operator [_]_ is itself defined in an ELAN module by a set of rewrite rules and a strategy `eval`.

2.3. *Rules involving strategies in conditions*

From now on, let us consider that not only rules but also strategies can be applied on terms. Moreover a rule itself may call a strategy in its matching conditions. So the general form of a rule in ELAN is

$$[\mathsf{lab}]\ l \to r\ \mathbf{where}\ p_1 := \{S_1\}c_1\ \ldots\ \mathbf{where}\ p_n := \{S_n\}c_n$$

in which

— $l, r, p_1, \ldots, p_n, c_1, \ldots, c_n \in \mathcal{T}(\mathcal{F}, \mathcal{X})$,

— $\mathcal{V}ar(p_i) \cap (\mathcal{V}ar(l) \cup \mathcal{V}ar(p_1) \cup \cdots \cup \mathcal{V}ar(p_{i-1})) = \emptyset$,

— $\mathcal{V}ar(r) \subseteq \mathcal{V}ar(l) \cup \mathcal{V}ar(p_1) \cup \cdots \cup \mathcal{V}ar(p_n)$,

— $\mathcal{V}ar(c_i) \subseteq \mathcal{V}ar(l) \cup \mathcal{V}ar(p_1) \cup \cdots \cup \mathcal{V}ar(p_{i-1})$,

— $S_1, \ldots, S_n$ are strategy terms built from the strategy constructors or defined by the user,

— and $\{\_\}\_$ is either the application operator $(\_)\_$ of elementary strategies on terms, or the application operator $[\_]\_$ of defined strategies on terms.

When programming in ELAN, it is important to realize the difference between:

– **labelled rules** whose evaluation is fully controled by the user strategies and,

– **non-labelled rules** which are intended to perform deterministic computations and are applied using a built-in leftmost innermost strategy.

Rules and strategies provide a very flexible language paradigm for modelling and prototyping many applications: decision procedures, theorem provers, constraint solvers have been prototyped in ELAN. Solving techniques for constraint satisfaction problems (CSP for short) over finite domains have been implemented in the system COLETTE [CAS 98a, CAS 98b] written in ELAN. Actually, rules are also natural to describe planification problems, and to decompose complex tasks into primitive actions, according to resources constraints. The second part of the paper develops an example, designed in ELAN and using COLETTE as a sub-program, of modelling a print controller.

3. Modelling a print controller

A print controller gives print tasks to execute to printers. The general aim is to distribute as well as possible all print tasks on the network and to assure that all print tasks are achieved before a given deadline. The general problem considers several printers connected to a server by lines. The two first kinds of resources are the number of printers and the number of lines available. There may be more than one printer connected by a line. The third resource is the number of memories.

We base our specification on a specification given in [AND 98] where the problem of minimizing the deadline and the resourses management problem are not addressed. Here, we made similar restrictions on the general problem.

Since we want here to focus our attention on modelling the control management, we simplify the problem by considering only one printer, with one line and enough memory. Due to this simplification, the constraints associated to this problem are only time constraints, expressing that a task begins after another one ends.

There are two kinds of requests: *simple print tasks* and *complex print tasks*. A simple print task consists of printing once a document. The duration associated to this task is fixed and it cannot be interrupted. A complex print task consists of loading in memory the document to print, of printing it several times and when finished, of

freeing the allocated memory space. This complex print task can be interrupted but, in this case, the allocated memory space must be released and when starting again this job, the document has to be reloaded.

In our running example, the purpose is to find possible schedules for executing on the printer a simple print task and a complex one composed of N print tasks. The different possibilities are either to split the complex print task or to perform it in one step before or after the simple print task. Moreover a deadline is fixed and given as a data of the problem: the purpose is not to minimize the execution time for all these requests, but to find all possible schedulings within the given deadline.

For instance, let us consider a request composed of a simple print task, taking 2 time units, and a complex task composed of 2 print tasks, taking 1 time unit each, on one printer, with a deadline of 6 time units. Assuming no time is left between two tasks, the three possible schedules are presented in Figure 1 where SP denotes the simple print task. A complex task always begins by Loading (L) the job following by one or more Print (P) tasks. The FormKeep (FK) task ensures that the document is still loaded during the execution.

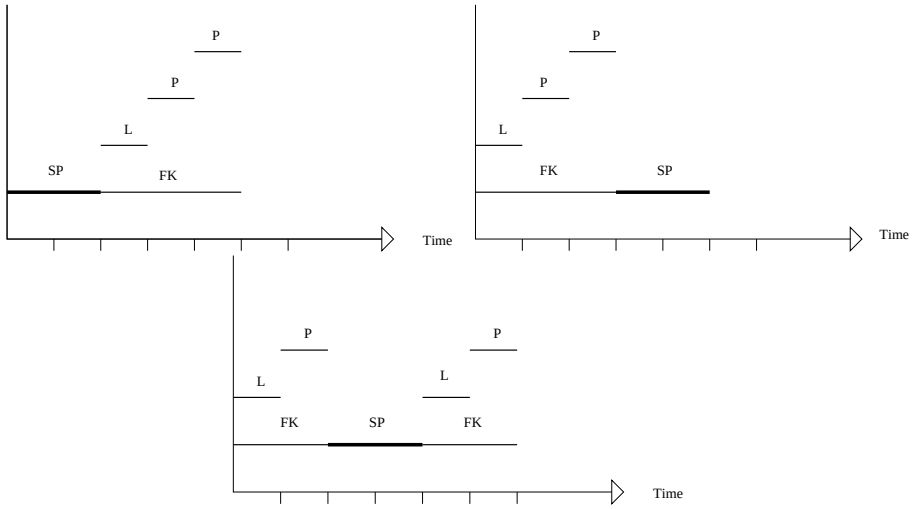

Figure 1. *Three Execution Schedules*

This problem was formalized in [AND 98] with grammar rules and feature terms, i.e. structures given by a set of attributes/value pairs. This first formalization, with slightly modified notations, is presented in Figure 2. Five rules decompose complex tasks into other complex tasks or into primitive actions. Tasks have three attributes giving respectively:

— their name,

— the number of impressions that the task must manage,

— and the status for the decomposition of this task, either all or $split$.

Actions have two attributes giving respectively:

— their name,

— and the duration of the action.

The general structure of these rules is $LS \rightarrow RS$ w.r.t. a constraint C. LS is a feature term that represents a complex task. RS is a list of feature terms corresponding to newly generated complex tasks and primitive actions.

The first rule specifies the different tasks to be scheduled: here the task Main (denoted M) is composed of a complex task FormPrint (FP) to perform, and of an primitive action SimplePrint (SP).

The two following rules are for decomposing the complex task FormPrint (FP). There are two possible choices here: either we do not decompose this task, but we load (L) the document, multi-print it N times, while we assure that the document is kept in memory during all these operations (this is the primitive action FormKeep (FK)). Or we decide to decompose the complex task into two parts. The initial number N of documents to print is split into $N1$ and $N2$ such that $N1 + N2 = N$.

The two last rules decompose the complex task MultiPrint (MP). It simply consists, for printing a document N times, in printing it once and then $N - 1$ times.

To summarize, complex tasks are M, FP and MP, primitives actions are SP, P, L and FK.

task = M	–>	task = FP	action = SP
			duration = 2

SP precedes FP or FP precedes SP or (FP begins before SP and ends after SP)

task = FP	–>	action = L	task = MP	action = FK
status = all		duration = 1	numI = N	
numI = N				

L precedes MP and FK begins with L and FK ends with MP

task = FP	–>	task = FP	task = FP
status = split		numI = N1	numI = N2
numI = N			

$N1 + N2 = N \ \wedge \ N1 \geq 0 \ \wedge \ N2 \geq 0$

| task = MP | numI = 1 | –> | action = P | duration = 1 |

| task = MP | numI = N | –> | action = P | duration = 1 | task = MP | numI = N-1 |

P precedes MP

Figure 2. *Formalisation of a print controller.*

Let us see now how to translate this formalisation into ELAN.

3.1. *The data structures*

Constraints, actions and tasks with their attributes are encoded into different structures. In order to formalise the scheduling constraints, we associate two variables to each complex task and primitive action named X: its beginning time (noted $X.B$), and its ending time ($X.E$). The duration $X.D$ relates these two values ($X.E = X.B + X.D$). The domain of each variable is initialised to the integer interval $[0..T]$ where T is the deadline.

The constraint associated to M in Figure 2 (SP precedes FP or FP precedes SP or (FP begins before SP and ends after SP)) is expressed as:

$$FP.B = SP.E \vee SP.B = FP.E \vee (SP.B > FP.B \wedge SP.E < FP.E)$$

(either FP begins when SP ends, or SP begins when FP ends, or FP begins before SP and ends after SP).

The constraints associated to FP with status all (FK begins with L, L precedes MP and FK ends with MP) is written as:

$$L.B = FK.B \wedge L.E = MP.B \wedge MP.E = FK.E.$$

The constraints associated to FP with status $split$ require the following conditions: as soon as the N1 prints end, the N2 prints begin or conversely:

$$FP(N1).E = FP(N2).B \vee FP(N1).B = FP(N2).E).$$

A CSP is defined by a set of variables, their respective domains and a set of constraints built as disjunctions and conjunctions of equations, inequations and disequations on arithmetic expressions with addition ans substraction on finite domains. A CSP is represented in ELAN by a structure with five components: the list of membership constraints of the form $(X \in^? D)$ where X is a variable and D its domain, a list of equality constraints of the form $(X =^? V)$ where V is an arithmetic expression, a list of conjunctive constraints and a list of disjunctive constraints. The last component stores intermediate results.

Then, a primitive action is represented by a structure with three components: its name, its beginning and ending times: $< Name\ ABegin : AEnd >$.

A complex task is given by a structure with five components: its name, the number of current impressions not yet done, its status telling if we are in a $split$ mode or in an all mode, its beginning and ending times:

$$< \ Name \ NB-Imp \ Status \ TBegin \ : \ TEnd \ >.$$

To manipulate the entire structure of the problem, we define a universe of sort `state` including five components: the list of complex tasks to be decomposed, the list of primitive actions to execute, the current CSP problem to solve, the number of variables used in the CSP and in the definition of primitive actions and complex tasks and the time domain that defines boundaries for the values of the variables. For example, if the domain is `[0,..,3]`, each action and task must begin after the time `0` and end before `3`. So, the universe is defined as the following structure, where components are separated by $\|$:

$list-tasks \ \| \ list-actions \ \| \ CSP \ \| \ NBVars \ \| \ time-domain.$

The two first components are concerned with the decomposition of complex tasks into primitive actions, while the three others are dealing with the constraint satisfaction problem and provide the interface between the rules and the constraints.

3.2. *The rewrite rules*

Rules given in Figure 2 are translated into ELAN rules.

The general structure of a rule includes five steps: the creation of new variables (`n1`, `n2`, `n3` and `n4`), the production of the constraints over the new tasks or actions (`Const`), the integration of this new constraint within the global CSP (`C1`), the satisfaction test of the CSP and the updating of the universe (`S`). We show the rule corresponding to the decomposition of the complex task `FormPrint` when the status is *all* (the second rule in Figure 2):

```
[F1] <FormPrint N all B : E>.LT || LA || C || NBVar || T
        => S
        where n1 := () NBVar+1
        where n2 := () NBVar+2
        where n3 := () NBVar+3
        where n4 := () NBVar+4
        where Const := () V_$n1=?B & V_$n2=?V_$n1(+)1 &
                          V_$n3=?V_$n2 & V_$n4=?E
        where C1 := (LocalConsistencyForEC) ComposeCSP &
                              Constraint(C,Const,n4,4,T)
        if satisfiable(C1)
        where S := () <MultiPrint N all V_$n3 : V_$n4>.LT ||
              <Load V_$n1 : V_$n2>.<FormKeep V_$n1 : V_$n4>.LA || C1 || n4 || T
end
```

The whole set of rules is given in Appendix.

3.3. *Constraint solving and satisfiability checking*

For solving constraint satisfaction problems in our example, we use the COLETTE system. In this section, we will briefly present the notions used for our study. For more results about CSP, the reader may refer to [CAS 98a]. Rather than testing COLETTE on this type of application, our intention was to see how easily it could be integrated into another ELAN program. As expected, the integration was quite easy,

just by adapting a few top-level primitives of COLETTE for our specification. We had to adapt the creation of the CSP and the initialization of the constraints on the domain (for this initialization, the last component of the structure `state` is needed).

To perform the integration of a new constraint in the current CSP, the new constraint is first transformed into a CSP and simplified (i.e. domains of variables are reduced by a local consistency technic), then composed with the current CSP and the result is simplified again.

COLETTE also provides a library of strategies for solving constraint satisfaction problems. Among them we have tried two of them: the Forward Checking strategy and the Full Look Ahead one. The results were quite equivalent for our example.

We give below some results obtained with the Forward Checking strategy with an enumeration of the results from left to right. This strategy is called in COLETTE `FCFirstToLast`. We have used this strategy in two ways. The first one is the entire solving of the CSP and the enumeration of all solutions. The second way just consists of stopping the enumeration as soon as the first solution is found: this mode is called the satisfiability mode. We give here the strategy `FCFirstToLast`:

```
[] FCFirstToLastAll2 =>
repeat* (
        dk (iterate* (EliminateFirstValueOfDomain)) ;
        first one (InstantiateFirstValueOfDomain) ;
        first one (ExtractConstraintsOnEqualityVar, id) ;
        first one (Elimination , id) ;
        first one (LocalConsistencyInEC , id)
        )
first one (GetSolutionCSP2)
end
```

3.4. *The choice points*

Several choice points are set either by the rules, or by the CSP solving process. These are places were backtracking is performed. Let us detail them:

— In the Main rule (the first one in Figure 2), either we call the complex task with the status all, or with the status $split$.

— In the FormPrint rule with the status $split$ (the third rule in Figure 2), the first choice point is set when splitting in two the number of print tasks. We can decompose 4 print tasks in 3 ones and then 1, but also in 2 ones and then 2 other ones. This choice point is given by a strategy:

```
[] Split        => dk(Splitting)
```

where `Splitting` is the label of a rule enumerating the different splittings.

The second choice point in this rule is set when we decide the status of the newly generated FormPrint tasks. These new tasks can have the status all by applying a rule labelled `ST1`, or status $split$ by applying `ST2`. This is done by the strategy:

```
[] ChooseStatus => dk(ST1 , ST2)
```

All these choice points correspond to branching in the derivation tree for the decomposition of complex tasks into primitive actions. Moreover, in each leaf of the derivation tree, when COLETTE is called for the complete solving of the CSP, we can have different solutions, and so a lot of choice points generated by COLETTE.

3.5. *Strategies for the rules composition*

The main rule for the decomposition of the complex task Main (the first one in Figure 2) is translated in ELAN into three rewrite rules labelled by `Main1`, `Main2` and `Main3`, each one corresponding to a disjunct in the constraint of Figure 2. The strategy `Main` is defined as a non-deterministic choice of these three ELAN rules.

The FormPrint complex task is also defined by three rules in ELAN labelled `F1` for FP with status all, `F2` and `F3` for FP with status $split$, again each one corresponding to a disjunct in the constraint. The `Form` associated strategy is again a non-deterministic choice of these three ELAN rules.

For the MultiPrint task, we have also two rules in ELAN labelled `MP1` (for one print) and `MP2` (recursively rule for n prints) and the associated strategy is called `Multi`.

These three strategies are defined as follows:

```
[] Main    => dk (Main1 , Main2 , Main3)
[] Form    => dk (F1 , F2 , F3)
[] Multi   => first (MP1 , MP2)
```

3.6. *The evaluation mechanism*

The construction of the search tree can be done with two execution modes: a Full mode and a Step-by-Step mode.

— The Full mode uses a strategy `AllTogether` that develops all branches of the tree, checking at each node that the constraint is satisfiable. At each leaf, we get a possible decomposition for the main task into primitive actions and we solve the CSP to find all possible values for the variables. The strategy `AllTogether` calls the `Main`, `Form`, `Multi` strategies defined above, and a strategy `Solutions` that solves the CSP.

```
[]AllTogether=>Main;repeat*(Form;repeat*(Multi));Solutions
```

For the example illustrated in Figure 1, we find the three pictured schedules. For another request composed of a simple print task, taking 2 time units, and a complex task composed of 3 print tasks, taking 1 time unit each, on one printer, with a deadline of 8 time units, we find twelve possible schedules. We can notice that in the next table, the number of rewriting steps includes all rewriting steps done by COLETTE for the CSP resolution and the CSP satisfiability tests. The number of steps done by the print

controller is just a part of the whole number.

Example	Schedules	Execution Time	Rewriting Steps
(1+2)	3	0.11s	191 972
(1+3)	12	0.2s	8 698 593

— The Step-by-Step mode guides the user during the development of a solution, with a menu presented on the screen:

```
Could you give us the number associated to the strategy
you want now to execute (terminated by 'end')?:
1- Main
2- FormPrint
3- MultiPrint
4- All Results
5- One Result
6- Cut this branch
```

The three first choices help the user to develop the tree and to eliminate all complex tasks from the list of tasks. The fourth and the fifth ones guide the application of the COLETTE strategy: it gives either all results, or only the first one (as for the test of satisfiability). The sixth choice allows the user to cut the current branch of the tree, and the exploration starts again at the last set choice point.

The user choice is guided by the display of the current situation, as shown bellow:

```
 List of Tasks : <MultiPrint 1 all V_21:V_22>.nil
 List of Actions : <Load V_19:V_20>.<FormKeep V_19:V_22>.
<Print V_17:V_18>.<Print V_13:V_14>.<Load V_9:V_10>.
<FormKeep V_9:V_12>.<SimplePrint V_3:V_4>.nil
```

As an example, from the query `Step(<Main 4 split 0:8>)`, one can reach the following result:

```
<Print 7:8>.<Load 6:7>.<FormKeep 6:8>.
<Print 3:4>.<Load 2:3>.<FormKeep 2:4>.
<Print 1:2>.<Load 0:1>.<FormKeep 0:2>.
<SimplePrint 4:6>.nil
```

This schedule proposes to execute the simple print task SimplePrint between time 4 and 6. The complex print task is decomposed into three ones that are executed for the first one between 0 and 2 (Load between 0 and 1 - Print between 1 and 2 - FormKeep between 0 and 2), for the second one between 2 and 4 (Load between 2 and 3 - Print between 3 and 4 - FormKeep between 2 and 4) and for the last one between 6 and 8 (Load between 6 and 7 - Print between 7 and 8 - FormKeep between 6 and 8).

4. Conclusion

Several advantages of this modelling with rules and strategies can be emphasized: this provides the user with a high-level programming language for expressing com-

plex tasks and their decomposition into primitive actions. In the same time, it is easy to change the definition of strategies and rules. Then, thanks to the ELAN rewrite engine and compiler, one gets easy and fast prototyping of an application. Last, but not least, this declarative formalisation of tasks scheduling problems provides an adequate framework for the verification of properties of rules. For rules not involving strategies in their conditions, one can check for instance confluence and termination, absence of contradiction, or covering properties. However much work remains to do for proving properties of strategies.

Aknowledgments:
We sincerely thank Jean-Marc Andréoli and Stefania Castellani for very interesting discussions and for their help in understanding this application. This work has been partially supported by the Esprit Basic Research Working Group 22457 - Construction of Computational Logics II.

Bibliographie

[AND 98] ANDREOLI J., BORGHOFF U., CASTELLANI S., PARESCHI R. et TEEGE G., « Agent Based Decision Support for Managing Print Tasks ». In *Proceedings of the PAAM'98, London, UK*, 1998.

[BAA 98] BAADER F. et NIPKOW T., *Term Rewriting and all That*. Cambridge University Press, 1998.

[BOR 96] BOROVANSKÝ P., KIRCHNER C. et KIRCHNER H., « Controlling Rewriting by Rewriting ». In MESEGUER J., Ed., *Proceedings of the first international workshop on rewriting logic*, vol. 4, Asilomar (California), September 1996. Electronic Notes in Theoretical Computer Science.

[BOR 98] BOROVANSKÝ P., KIRCHNER C. et KIRCHNER H., « A functional view of rewriting and strategies for a semantics of ELAN ». In SATO M. et TOYAMA Y., Eds., *The Third Fuji International Symposium on Functional and Logic Programming*, p. 143–167, Kyoto, April 1998. World Scientifi c. Also report LORIA 98-R-165.

[CAS 98a] CASTRO C., « *Une approche déductive de la résolution de problèmes de satisfaction de contraintes* ». PhD thesis, Université Henri Poincaré - Nancy I, 1998.

[CAS 98b] CASTRO C., « Building Constraint Satisfaction Problem Solvers Using Rewrite Rules and Strategies ». *Fundamenta Informaticae*, vol. 34, p. 263–293, September 1998.

[CIR 97] CIRSTEA H. et KIRCHNER C., *Theorem proving using computational systems: The case of the B Predicate prover*. http://www.loria.fr/cirstea/Papers/PredicateProver.ps, 1997.

[DER 90] DERSHOWITZ N. et JOUANNAUD J.-P., « *Handbook of Theoretical Computer Science* », vol. B, Chapitre 6: Rewrite Systems, p. 244–320. Elsevier Science Publishers B. V. (North-Holland), 1990. Also as: Research report 478, LRI.

[KIR 95a] KIRCHNER C., KIRCHNER H. et VITTEK M., Designing Constraint Logic Programming Languages using Computational Systems. In VAN HENTENRYCK P. et SARASWAT V., Eds., *Principles and Practice of Constraint Programming. The Newport Papers.*, Chapitre 8, p. 131–158. The MIT press, 1995.

[KIR 95b] KIRCHNER H. et MOREAU P.-E., « Prototyping completion with constraints using computational systems ». In HSIANG J., Ed., *Proceedings 6th Conference on Rewriting Techniques and Applications, Kaiserslautern (Germany)*, vol. 914 de *Lecture Notes in Computer Science*, p. 438–443. Springer-Verlag, 1995.

[KIR 98] KIRCHNER C. et RINGEISSEN C., « Rule-Based Constraint Programming ». *Fundamenta Informaticae*, vol. 34, n° 3, p. 225–262, September 1998.

[KLI 93] KLINT P., « A meta-environment for generating programming environments ». *ACM Transactions on Software Engineering and Methodology*, vol. 2, p. 176–201, 1993.

[MES 92] MESEGUER J., « Conditional rewriting logic as a unified model of concurrency ». *Theoretical Computer Science*, vol. 96, n° 1, p. 73–155, 1992.

[RIN 97] RINGEISSEN C., « Prototyping Combination of Unification Algorithms with the ELAN Rule-Based Programming Language ». In *Proceedings 8th Conference on Rewriting Techniques and Applications, Sitges (Spain)*, vol. 1232 de *Lecture Notes in Computer Science*, p. 323–326. Springer-Verlag, 1997.

Appendix: ELAN rules

```
rules for state

        N,N1,N2,NBVar   : int;
        st,st1,st2      : status;
        B,E,t1,t2       : term;
        LT              : list[task];
        LA              : list[action];
        C,C1            : csp;
        n1,n2,n3,n4     : int;
        Const           : constraint;
        S               : state;
        T               : domain;

global

// 1st case: FP (within status all) is done after SP (cf constraint V_$n4=?V_$n1)
[Main1] <Main N st B : E>.LG || LA || C || NBVar || T    => S
              where n1 := () NBVar+1
              where n2 := () NBVar+2
              where n3 := () NBVar+3
              where n4 := () NBVar+4
              where Const := () V_$n1>=?B & V_$n3=?B & V_$n2<=?E & V_$n4<=?E &
        V_$n4=?V_$n3(+)2 & V_$n4=?V_$n1
              where C1 := (LocalConsistencyForEC) ComposeCSP&Constraint(C,Const,n4,4,T)
              if satisfiable(C1)
              where S := () <FormPrint N-1 all V_$n1 : V_$n2>.LT ||
        <SimplePrint V_$n3 : V_$n4>.LA || C1 || n4 || T
end

// 2nd case: FP (within statut all) is done before SP (cf constraint: V_$n2=?V_$n3)
[Main2] <Main N st B : E>.LT || LA || C || NBVar || T      => S
              where n1 := () NBVar+1
              where n2 := () NBVar+2
              where n3 := () NBVar+3
              where n4 := () NBVar+4
              where Const := () V_$n1=?B & V_$n3>?B & V_$n2<=?E & V_$n4<=?E &
        V_$n4=?V_$n3(+)2 & V_$n2=?V_$n3
              where C1 := (LocalConsistencyForEC) ComposeCSP&Constraint(C,Const,n4,4,T)
              where C2 := () printnl(C1)
              where S := () <FormPrint N-1 all V_$n1 : V_$n2>.LT ||
        <SimplePrint V_$n3 : V_$n4>.LA || C1 || n4 || T
end

// 3rd case: SP is done between several FP (within the statut split)
//(cf constraint: V_$n3>?V_$n1 & V_$n4<?V_$n2)
[Main3] <Main N st B : E>.LT || LA || C || NBVar || T      => S
              where n1 := () NBVar+1
              where n2 := () NBVar+2
              where n3 := () NBVar+3
              where n4 := () NBVar+4
```

```
              where Const := () V_$n1>=?B & V_$n3>=?B & V_$n2<=?E & V_$n4<=?E &
       V_$n4=?V_$n3(+)2 & V_$n3>?V_$n1 & V_$n4<?V_$n2
              where C1 := (LocalConsistencyForEC) ComposeCSP&Constraint(C,Const,n4,4,T)
              if satisfiable(C1)
              where S := () <FormPrint N-1 split V_$n1 : V_$n2>.LT ||
       <SimplePrint V_$n3 : V_$n4>.LA || C1 || n4 || T
end

////////////////////////////////    Rules for FORMPRINT

// 1st case: the statut is all. We decompose into actions L and FK and into complex task MP
[F1] <FormPrint N all B : E>.LT || LA || C || NBVar || T       => S
              where n1 := () NBVar+1
              where n2 := () NBVar+2
              where n3 := () NBVar+3
              where n4 := () NBVar+4
              where (pair[term,term]) [t1,t2] := () GetTimesTask(SimplePrint,LA)
              where Const := () V_$n1=?B & V_$n2=?V_$n1(+)1 & V_$n3=?V_$n2 &
       V_$n4=?E & t1>=?V_$n4 V t2<=?V_$n1
              where C1 := (LocalConsistencyForEC) ComposeCSP&Constraint(C,Const,n4,4,T)
              if satisfiable(C1)
              where S := () <MultiPrint N all V_$n3 : V_$n4>.LT ||
       <Load V_$n1 : V_$n2>.<FormKeep V_$n1 : V_$n4>.LA || C1 || n4 || T
end

// 2nd case: th statut is split. We decompose into 2 FP. For the two generated FP, the 1st
// one is done before the 2nd (following the list of tasks - cf constraint: V_$n2<=?V_$n3).
[F2] <FormPrint N split B : E>.LT || LA || C || NBVar || T      => S
              if N > 1
              where n1 := () NBVar+1
              where n2 := () NBVar+2
              where n3 := () NBVar+3
              where n4 := () NBVar+4
              where (pair[int,int]) [N1,N2] := (Split) split(N)
              where Const := () V_$n1=?B & V_$n3>?B & V_$n2<=?V_$n3 & E=?V_$n4
              where C1 := (LocalConsistencyForEC) ComposeCSP&Constraint(C,Const,n4,4,T)
              if satisfiable(C1)
              where st1 := (ChooseStatus) Stat
              if not (N1==1 and st1==split)
              where st2 := (ChooseStatus) Stat
              if not (N2==1 and st2==split)
              where S:= ()<FormPrint N1 st1 V_$n1:V_$n2>.<FormPrint N2 st2 V_$n3:V_$n4>.LT
       || LA || C1 || n4 || T
end

// 3rd case: idem above. We just inverse the order for the tasks: the 2nd is done
// before the 1st (cf constraint: V_$n4<=?V_$n1).
[F3] <FormPrint N split B : E>.LT || LA || C || NBVar || T      => S
              if N > 1
              where n1 := () NBVar+1
              where n2 := () NBVar+2
              where n3 := () NBVar+3
              where n4 := () NBVar+4
              where (pair[int,int]) [N1,N2] := (Split) split(N)
              where Const := () V_$n1>?B & V_$n3=?B & V_$n4<=?V_$n1 & E=?V_$n2
              where C1 := (LocalConsistencyForEC) ComposeCSP&Constraint(C,Const,n4,4,T)
              if satisfiable(C1)
              where st1 := (ChooseStatus) Stat
              if not (N1==1 and st1==split)
              where st2 := (ChooseStatus) Stat
              if not (N2==1 and st2==split)
              where S:=()<FormPrint N1 st1 V_$n1:V_$n2>.<FormPrint N2 st2 V_$n3:V_$n4>.LT
|| LA || C1 || n4 || T
end

////////////////////////////////    Rules for MULTIPRINT
```

```
// 1st case: we've got only one action for printing.
[MP1] <MultiPrint 1 st B : E>.LT || LA || C || NBVar || T       => S
              where n1 := () NBVar+1
              where n2 := () NBVar+2
              where Const := () V_$n1=?B & V_$n2=?V_$n1(+)1 & E=?V_$n2
              where C1 := (LocalConsistencyForEC) ComposeCSP&Constraint(C,Const,n2,2,T)
              if satisfiable(C1)
              where S := () LT || <Print V_$n1 : V_$n2>.LA || C1 || n2 || T
end

// 2nd case: we have to print the job N times. Firstly 1, and then (N-1) times.
[MP2] <MultiPrint N st B : E>.LT || LA || C || NBVar || T       => S
              if N > 1
              where n1 := () NBVar+1
              where n2 := () NBVar+2
              where n3 := () NBVar+3
              where n4 := () NBVar+4
              where Const := () V_$n1=?B & V_$n2=?V_$n1(+)1 & V_$n2=?V_$n3 & E=?V_$n4
              where C1 := (LocalConsistencyForEC) ComposeCSP&Constraint(C,Const,n4,4,T)
              if satisfiable(C1)
              where S := () <MultiPrint N-1 st V_$n3 : V_$n4>.LT ||
       <Print V_$n1 : V_$n2>.LA || C1 || n4 || T
end
end
```