Actions & Plans in ELAN

Hubert Dubois, Hélène Kirchner

Actions and plans in ELAN

Hubert Dubois and Hélène Kirchner
LORIA-UHP & CNRS
BP 239
54506 Vandœuvre-lès-Nancy Cedex, France

Abstract. ELAN is a declarative language based on rewriting logic. The ELAN language is based on labelled conditional rewrite rules and on strategies for controlling their application. ELAN provides a strategy language to control labelled rules. In this paper we show how to use the ELAN strategy language for planning. We describe how to encode situations and actions and take advantage of ELAN strategies to build plans. We give an example of our approach by describing an elevator controller.

1 Introduction

Among numerous contributions in the area of logic and changes, rewriting logic has been proposed in [MOM94] as a logic of concurrent action and change that solves the frame problem and unifies a number of previous logics of change including linear logic and Horn logic with equality.
ELAN is a declarative language based on rewriting logic [Mes92] and its logical foundations are detailed in [KKV95,BKK96].
In ELAN a rewrite rule may be labelled, may have conditions and may introduce local variables useful to compute intermediate results. A computation may have several results which introduces the need to deal with non-determinism and backtracking.
One of the main originality of the language is to provide strategy constructions to specify whether a function call returns several, at least one or at most 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 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.
ELAN has been used as an environment for specifying and prototyping constraint solvers, theorem provers and deduction systems in general. It also provides a framework for experimenting their combination.
The current version of ELAN includes an interpreter and a compiler written 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 [Cas97], theorem proving tools in first-order logic with equality [KM95,CK97], the combination of unification algorithms and of decision procedures in various equational theories [Rin97]. More information on the system can be found on the WEB site[1].
We open in this yet prospective paper a new application area for the ELAN language. We want to experiment its ability to model planning problems. A correspondence is drawn between actions and strategies, where primitive actions correspond to primal strategy (i.e. labelled rewrite rules) in ELAN, and where more complex actions are expressed using the full power of the strategy language. Then plans are a specific form of proof terms generated by the execution of a complex action on an initial situation. We mention in conclusion a few points for further research perspectives.

2 ELAN rules and strategies

An ELAN program describes a set of operators with possibly structural axioms (like associativity and commutativity properties), a set of conditional rewrite rules that can be named by labels, a set of strategy operators and a set of strategy rules.
In ELAN, rules are labelled conditional rewrite rules with local variable assignments

$$[\mathsf{lab}] : l \Rightarrow r \ \textbf{if}\ v \ \textbf{where}\ y := (S)u$$

where lab is the label, l and r the respective left and right-hand sides, v the condition and $y := (S)u$ a local assignment, giving to the local variable y the results of the strategy S applied to the term u. Any sequence of **where** and **if** is allowed but their order is relevant for the evaluation. For applying such a rule on a term t, say at top position, first l is matched against t, then the expressions introduced by **where** and **if** are instantiated with the matching substitution and evaluated in order. Instantiations of local variables (such as y) after **where** extend the matching substitution. When every condition is satisfied, the replacement by the instantiated right-hand side is performed.
Unlabelled rules are applied with a leftmost-innermost strategy by the interpreter. On the contrary, labelled rules are used in user-defined strategies and applied at top position in the term to be reduced. Instantiations of local variables after **where** also invoke ELAN strategies, built from a few constructors. Let us explain now how to express a strategy.

- A labelled rule is a primal strategy. The result of applying a rule labelled lab on a term t returns a set of terms.
- **first-one**$(S_1, \ldots, S_n)$ chooses the first strategy S_i in the list that does not fail, and returns its first result. This strategy is said deterministic since it returns at most one result.
- **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 is said non-deterministic since it may fail or return more than one one result.

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

- **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 strategy is again non-deterministic.
- $S_1; S_2$ is 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.
- **id** is the identity that does nothing but does not fail.
- **fail** is the strategy that always fails.

The strategy language and its semantics are described in [BKK98], thanks to an application operator

$$[\_](\_) : (Strategy, Term) \quad SetOfTerms$$

whose interpretation is given by labelled rewrite rules.
ELAN also gives the possibility to the user to define recursive and parameterised strategies with rewrite rules [BKK96,BKK97]. Given a set of strategy symbols, rewrite rules on user-defined strategies are of the form [lab] $S_1 \Rightarrow S_2$ where lab is a label, S_1 and S_2 are strategy terms built from all previously introduced strategy symbols. Such strategy rewrite rules are called *implicit* since they do not involve explicitly the application operator @. However, it is sometimes useful to express a strategy rule depending on the argument on which it is applied. Such rules are also allowed and are called *explicit* strategy rules. They are of the form:

$$[\mathsf{lab}][S_1](t) \Rightarrow t'$$

where S_1 is a strategy term as before, t is a term and t' is built on function and strategy symbols and possibly the strategy application operator. From the evaluation point of view, these rules are just added to the strategy interpreter.

3 Situations, actions and changes

In order to model action and change, we use a formalism very similar to the situation calculus. This first-order language is devoted to represent dynamically changing worlds. First introduced at the end of the 60' by Hayes and McCarthy in [MH69], the situation calculus was rediscovered in the earlier 90' in some recent works like [GLR91].
A *situation* is a first-order term representing a possible world history, which records changes to the world resulting from actions. Here a situation is defined by a pair of a state and a sequence of *primitive actions*. The initial situation S_0 is characterised by an initial state and an empty sequence of actions. The situation calculus introduces a reserved binary function symbol *do*, that builds from an action α and a situation s, a new situation $do(\alpha, s)$. For instance, consider an elevator controller. For the action $\alpha = up(5)$ (we want to go up to floor 5), $do(up(5), s)$ is the situation resulting from the application of $up(5)$ on the situation s.
A world dynamically changes during the execution. For every action, some preconditions characterise when this action can be performed. For instance, we only can move a block c from the place A to the place B if we

carry the block c. This precondition may depend on the current situation. So-called *fluents* are used to test or evaluate changes. Relational fluents return a boolean value, while others are called functional fluents.
For example, the relational fluent *is_carrying(robot,p,s)* asks if in the situation s the robot *robot* is carrying the object p; the results can be *true* or *false*. The functional fluent *location(robot,s)* indicates the location of the robot *robot* in the situation s; the results are not boolean.
The situation calculus introduces *frame axioms* in order to specify the unchanged fluents, during the application of a given action. The problem of defining these axioms is the huge number of them, although an action only changes a few of them. A solution to the frame problem has been described by Levesque and al. in [LRL+97] using a single axiom called *successor state axiom*.
Therefore a problem in the situation calculus is described by a set of axioms to define the initial situation, a successor state axiom for every fluent, axioms describing the primitives actions and a set of preconditions for every action.
Beyond primitives actions, more complex actions such as procedures, conditional actions, loops can be expressed. Such extensions are provided for instance in the language GOLOG [LRL+97]. So, an action is either a primitive action as before, or a complex action of the form:

- $[\mathbf{a_1}, ..., \mathbf{a_n}]$ for the sequence of actions $a_1, ..., a_n$.
- $?(\mathbf{p})$, where p is a *condition*. We call condition a relational fluent or an expression like $And(p_1, p_2)$, $Or(p_1, p_2)$, $Not(p)$ or $Some(v, p)$ where p, p_1 and p_2 are conditions and v is a variable that occurs in p and which is instantiated by a value taken from a set computed by a functional fluent. The action $?(\mathbf{p})$ stands for the test of the condition p.
- $\mathbf{a_1 \# a_2}$ for the non-deterministic choice between two actions a_1 and a_2.
- $\mathbf{if(p, a_1, a_2)}$ is a test of the condition p. If this test leads to *true*, then the action a_1 is performed, otherwise, a_2 is executed.
- $\mathbf{star(a)}$ repeats the execution of the action a ad infinitum.
- $\mathbf{while(p, a)}$ executes the action a as long as the condition p is *true*.
- $\mathbf{pi(v, a)}$, where v is a variable and a an action using v, assigns a value taken from a given set to the variable v and then instantiates this value by the action a.
- a procedure call, where a procedure is denoted by $\mathbf{proc(name, body)}$ where *name* is the identifier associated to the procedure name and the body of the procedure is the second argument of *proc*.

4 ELAN, actions and plans

When looking at the language of actions, it is natural to draw an analogy between primitive actions and primal strategy: applying an action is like applying in ELAN a labelled rewrite rule on a term representing a state (or situation). More complex actions need the full expressivity of the strategy language.
In a situation calculus application, thanks to the predicate *do*, the primitive actions are memorised in a situation. For instance, the situation

do(close,do(open,do(turnoff(5),do(up(5),S_0)))) indicates that, from the initial situation S_0, we went up to the floor 5, then we removed the floor 5 from the list of floors, opened the doors and finally close them. A *plan* is a succession of actions applied to the initial situation.
In a similar way, in ELAN at every step of rewriting, it is possible to record the label of the applied strategy. This correspond to the notion of proof term in rewriting logic initially defined in [MOM93] and extended to built-in strategies of ELAN in [BKK97].
Then plans are a specific form of proof terms generated by the execution of a complex action on an initial state. These proof terms are obtained by concatenation of primitive actions. Since in general we do not want to keep all the information involved in a rewriting step (i.e. the rule, the application position, the substitution), we use a restricted notion of proof terms that only memorises the names of primitive actions and their parameters.
Our ELAN term to reduce (the request) is then a pair composed of an object designing the current state and a proof term corresponding to the actions that have been executed to reach this state.

4.1 Specification of the situation

Let us consider the example of an elevator controller. The problem can be described by two components: first, the list of all floors to be served, and second, the current floor. In ELAN, we introduce a structure

```
<Liste-of-floors: L & Current-floor: CF>
```

with two attributes, list of floors and current floor, and a constructor `<@ & @>` which takes two arguments of respective sorts `list[int]` and `int`, and builds a result of sort `state`.

```
sorts int, list[int];
<@ & @> : (Liste-of-floors:list[int] Current-floor:int) state
```

This automatically provides two selectors:

```
@.Liste-of-floors : (state) list[int]
@.Current-floor   : (state) int
```

and some rules to extract and replace the attributes:

```
<L & C>.Liste-of-floors => L
<L & C>.Current-floor   => C
<L & C>[.Liste-of-floors <- L1] => <L1 & C>
<L & C>[.Current-floor <- C1]   => <L & C1>
```

Once the state is defined, we define a situation by a pair made of a state and a strategy term to represent the sequence of actions that have been performed to reach this state. We use the notation `s * P` to denote this pair. P is of sort `proof` standing for `state -> state` which is the sort of the strategies on the sort `state`. A term of the form `<L & CF> * P` is of sort `situation`.
The initial situation is given by the user at the beginning of the rewrite process. It describes the initial values of the list `L` and the current floor `CF`. For instance, if we start on floor 4 and the 3 and 5 are switched on, the initial state is `<3,5,nil & 4>`, and the initial sequence of actions is the empty list `nil`. The initial situation corresponds to the term `<3,5,nil & 4> * nil`.

4.2 The tests

In order to test if some condition is satisfied in a given state, a few rules are added. These rules do not modify the state when they apply, but fail when the test is not true.
In the elevator controller, we define `?(on(n))` and `?(current_floor(n))`, where `n` is an integer, for a given state `<L & CF>`. `?(on(n))` checks whether the floor `n` occurs in the list `L` of the floors not yet served; `?(current_floor(n))` checks whether the current floor `CF` is equal to `n`.

```
strategies for state
        n,CF    : int;
        L       : list[int];
explicit
[] [?(on(n))] <L & CF>                => <L & CF>
                if belongs(n,L)
end
[.] [?(current_floor(CF))] <L & CF>   => <L & CF>
end
end
```

Other rewrite rules model functional fluents depending on the current state. In our example, there are two functional fluents defined by rewriting rules: `next_floor` and `Set_Values`.

- `next_floor` computes which floor will be served next by considering the current state.
- `Set_Values` also depends on the current state. This set is composed of all possible values for variables that have to be instantiated. For each state, `Set_Values` is the union of the list of floors to be served and the current floor.

More complex tests can use connectors like `And`, `Or`, `Not` or `Some`. Such compound conditions are decomposed into primitive ones, using again rewrite rules, as shown below. These strategy definitions are given in a module parameterised by a general sort `X`, which introduces sorts `atom` and `condition`. They are applied with a pre-defined strategy in the strategy interpreter, which is denoted by the label `[.]`.

```
strategies for X
        x,y             : atom;
        T               : X;
        C,C1,C2         : condition;
explicit
[.] [?(Some(x,C))] T    => [?(sub_C(x,y,C))] T
        where y := (listExtract) elem(Set_Values(T))
end
implicit
[.] ?(And(C1,C2))        => ?(C1) ; ?(C2)
end
[.] ?(Or(C1,C2))         => ?(C1) # ?(C2)
end
[.] ?(Not(C))            => first(?(C) ; fail , id)
end
end
```

4.3 The actions

The strategy language is used to define primitive actions including their preconditions directly thanks to the `if` construction in the rule definition of a strategy in ELAN.
The strategy operators are the names of the primitive actions. The signature of strategies corresponding to primitive actions in our elevator controller example is given below. Note that strategies may have a parameter (`up` for instance).

```
stratop global
        turnoff(@)              : (int) <state>;
        open                    : <state>;
        close                   : <state>;
        up(@)                   : (int) <state>;
        down(@)                 : (int) <state>;
end
```

The possible primitive actions are the opening/closure of the doors by `open` and `close`, the fact of going up/down to a floor `N` by the actions `up(N)` and `down(N)` and the removal of a floor `N` from the list of floors by `turnoff(N)`. The rewrite semantics of these actions is given by a set of rewrite rules given below. Notice that two primitive actions have preconditions, namely `up` and `down`: we can only go up if the current floor is below the next floor (and the precondition is the opposite for `down`). Note that each application of a primitive action generates the concatenation of the corresponding strategy in the proof term.

```
strategies for state
        N,CF    : int;
        L,L1    : list[int];
        P       : list[proof];
explicit
[.] [turnoff(N)] <L & CF> * P => <remove(N,L) & CF> * turnoff(N),P
end
[.] [open] <L & CF> * P       => <L & CF> * open,P
end
[.] [close] <L & CF> * P      => <L & CF> * close,P
end
[.] [up(N)] <L & CF> * P      => <L & N> * up(N),P
        if CF < N
end
[.] [down(N)] <L & CF> * P    => <L & N> * down(N),P
        if CF > N
end
end
```

More complex action definitions use the full strategy language of ELAN. For instance, we easily define by simple rules on strategies, the non-deterministic choice between two actions `S1#S2`, the conditional `if(p,S1,S2)`, the loop `star(S)`, the iteration `while(p,S)` of `S` until condition `p` is false, the assignment `pi(v,S)` of a variable `v` that occurs in a strategy `S`.

```
strategies for X
        x,y             : atom;
        T               : X;
        C               : condition;
        S,S1,S2         : <X->X>;

explicit
[.] [pi(x,S)] T         => [sub_S(x,y,S)] T
        where y := (listExtract) elem(Set_Values(T))
end

implicit
[.] S1 # S2              => dk(S1,S2)
end
[.] if(C,S1,S2)          => first(?(C) ; S1 , S2)
end
[.] star(S)              => first(S ; star(S) , id)
end
[.] while(C,S)           => star(?(C) ; S)
end
end
```

The procedure call can be expressed by an implicit or explicit strategy rule of one of the following forms:

```
[.]  proc_name    => proc
end
[.] [proc_name]s  => s'
end
```

For instance, in the example of the elevator controller, we give the next procedures, among which `control` is the main one.

```
explicit
[.] [serve_a_floor] s   => s1
                where s1 := [serve(next_floor(s))] s
end

implicit
[.] go_floor(N) => ?(current_floor(N)) # up(N) # down(N)
end
[.] serve(N)    => go_floor(N) ; turnoff(N) ; open ; close
end
[.] park        => if(current_floor(0) , open , down(0) ; open)
end
[.] control     => while(Some(v,on(v)),serve_a_floor) ; park
end
end
```

4.4 The evaluation process

To execute this program, we give a request of the form `[control] S0` where `S0` is the initial term, for instance `[control] <3,5,nil & 4> * nil`. We get the following result:

```
<nil&0>*open,down(0),close,open,turnoff(3),down(3),close,open,
turnoff(5),up(5),nil
```

where the plan must be read from right to left.

5 Conclusion

This research is yet at a very preliminary stage. Our ultimate goal is to take advantage of this formulation of planning in rewriting logic to attack some planning and scheduling problems. Detecting inconsistent states, non-terminating or ambiguous plans should be possible in this formalisation. In addition it may be worth emphasising that, thanks to the concurrent nature of rewriting logic, plans are endowed with a notion of parallelism and we thus get for free a concurrent semantics for the execution of plans.

References

[BKK96] Peter Borovanský, Claude Kirchner, and Hélène Kirchner. Controlling rewriting by rewriting. In José Meseguer, editor, *Proceedings of the first international workshop on rewriting logic*, volume 4 of *Electronic Notes in Theoretical Computer Science*, Asilomar (California), September 1996.

[BKK97] P. Borovansky, C. Kirchner, and H. Kirchner. Strategies and rewriting in ELAN. In *Proc. of the CADE-14 workshop: Strategies in Automated Deduction*, Townsville, Australia, 1997. Report CRIN 97-R-126.

[BKK98] Peter Borovanský, Claude Kirchner, and Hélène Kirchner. A functional view of rewriting and strategies for a semantics of ELAN. In Masahiko Sato and Yoshihito Toyama, editors, *The Third Fuji International Symposium on Functional and Logic Programming*, pages 143–167, Kyoto, April 1998. World Scientific.

[Cas97] Carlos Castro. Constraint Manipulation using Rewrite Rules and Strategies. In Alice Drewery, Geert-Jan M. Kruijff, and Richard 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.

[CK97] Horatiu Cirstea and Claude Kirchner. *Theorem proving using computational systems: The case of the B Predicate prover.* http://www.loria.fr/cirstea/Papers/PredicateProver.ps, 1997.

[GLR91] M. Gelfond, V. Lifschitz, and A. Rabinov. What Are the Limitations of the Situation Calculus ? In R.S. Boyer, editor, *Automated Reasoning. Essays in Honor of Woody Bledsoe*, Automated Reasoning Series, chapter 8, pages 167–179. Kluwer Academic Publishers, Dordrecht, 1991.

[KKV95] Claude Kirchner, Hélène Kirchner, and Marian 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.*, chapter 8, pages 131–158. The MIT press, 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.

[LRL+97] Hector J. Levesque, Raymond Reiter, Yves Lespérance, Fangzhen Lin, and Richard B. Scherl. GOLOG : A Logic Programming Language for Dynamic Domains. *Journal of Logic Programming*, 31(1-3):59–83, 1997.

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

[MH69] J. McCarthy and P. Hayes. Some Philosophical Problems from the Standpoint of Artificial Intelligence. In B. Meltzer and D. Michie, editors, *Machine Intelligence 4*, pages 463–502. Edinburgh University Press, Edinburgh, Scotland, 1969.

[MOM93] N. Martì-Oliet and J. Meseguer. Rewriting logic as a logical and semantical framework. Technical report, SRI International, May 1993.

[MOM94] Narciso Marti-Oliet and Jose Meseguer. Action and change in rewriting logic. Technical Report SRI-CSL-94-07, SRI, April 1994.

[Rin97] Christophe 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.