Using Rewriting and Strategies for Describing the B Predicate Prover

Horatiu Cirstea, Claude Kirchner

Using Rewriting and Strategies for Describing the B Predicate Prover (Extended Abstract)

Horatiu Cirstea and Claude Kirchner

LORIA & INRIA
615, rue du Jardin Botanique, BP 101,
54602 Villers-lès-Nancy Cedex, France
email: {Horatiu.Cirstea,Claude.Kirchner}@loria.fr

Abstract. The framework of computational systems has been already used for describing several computational logics. In this paper is described the way a propositional prover and a predicate prover are implemented in ELAN, the system developed in Nancy for describing and executing computational systems. The inference rules for the provers are described by conditional rewrite rules and their application is controlled by strategies. We show how different strategies using the same set of rewrite rules can yield different proof methods.

1 Introduction

We start from the idea that logic programming in a broad sense, and theorem proving can be uniformly described in a logical framework. The rewriting logic ([MOM96]), a particular case of general logics ([Mes92]), provide such a logical framework in which many other logics can be represented by describing the syntax of formulas, the set of axioms, the set of deduction rules and the proof calculus.

The representation of a theorem prover like the predicate prover in this formalism allows to describe in a simple and expressive way the inference rules of the prover allowing on one hand the user to better understand its behaviour and on the other hand to prove properties of the rules. Furthermore, by making the formalisation executable we allow either to directly use the prover or to replay proofs done by another one.

In order to describe a computational version of a certain logic we use computational systems that can express the proof calculus of the given logic. A computational system ([KKV95]) is a combination of a rewrite theory and a strategy describing the intended set of computations. Computational systems are simple to understand as they are based on rewriting logic and first-order tools.

These ideas are implemented in the language ELAN ([BKK+96]) which allows to describe computational systems. In the environment provided by this language a logic can be specified by describing its syntax and inference rules. The inference

rules of the logic are described by conditional rewrite rules possibly containing local assignments. The application of these rules is controlled by strategies at two levels: the first one consists of defining expressions built over the alphabet of rule labels and the second one consists of using strategies in the local assignments of rule definitions.

We use ELAN as a logical framework for implementing the propositional calculus and the first-order predicate calculus whose syntax and inference rules are presented by J.-R. Abrial in [Abr96,Abr97].

The implementation of these rules in ELAN is done in a very natural way, most of the rewrite rules being syntactical transformations of the initial inference rules. ELAN provides a powerful definition of strategies used to guide the application of the rewrite rules and a great flexibility in developing new strategies. The description of the strategy is conceptually easy and the structure of the denoted proof terms can be straightforwardly derived from the computations. Thus, proofs of conjectures in the specified logics can be obtained by analysing the trace (provided in the ELAN environment) of the computations performed during the deduction process.

The section 2 presents the basic notions concerning the rewriting logic and the computational systems. The way these ideas are implemented in ELAN is then briefly presented. Section 3 presents some representative rules of the propositional and predicate prover and the strategies that guide their application. The last section of the paper contains the conclusions and gives further perspectives for this work. [1]

2 General Setting

This section presents the main concepts of rewriting logic that is fully described in [Mes92,MOM96]. The computational systems are then defined as rewrite theories in rewriting logic together with strategies that guide the computations. This provides the semantical foundation of the ELAN system as an environment in which computational systems can be described and executed.

2.1 Rewriting Logic

We just briefly recall the basic notions that are consistent with [JK91,DJ90] to which the reader is referred for a more detailed presentation. The definitions below are given in the one-sorted case, the many-sorted and order-sorted cases can be given a similar treatment.

We consider a set $\mathcal{F} = \bigcup_n \mathcal{F}_n$ of ranked function symbols, where $\mathcal{F}_n$ is the subset of functions of arity n, a set $\mathcal{X}$ of variables and the set of terms $\mathcal{T}(\mathcal{F}, \mathcal{X})$

[1] The implementations of the propositional and predicate prover can be tried at http://www.loria.fr/equipes/protheo/PROJECTS/ELAN/Applications/PropProver and http://www.loria.fr/equipes/protheo/PROJECTS/ELAN/Applications/Prover

built on $\mathcal{F}$ using the variables in $\mathcal{X}$. $\mathcal{T}(\mathcal{F},\mathcal{X})/E$ denotes the free algebra of terms modulo E, where E is a set of equalities.

In rewriting logic, proofs are first-order objects and are represented by proof terms. In order to build proof terms the rules are labelled with a set $\mathcal{L}$ of ranked label symbols. A *proof term* is by definition a term built on equivalence classes of $\mathcal{T}(\mathcal{F},\mathcal{X})/E$, function symbols from $\mathcal{F}$, label symbols from $\mathcal{L}$ and the concatenation operator ";". Thus, a proof term is an element of the algebra $\mathcal{PT} = \mathcal{T}(\mathcal{L} \cup \{;\} \cup \mathcal{F} \cup \mathcal{T}(\mathcal{F},\mathcal{X})/E)$.

A theory in this logic is a generalisation of the notion of theory from [Mes89], the axioms in the rewriting logic being identified by labels. A *labelled rewrite theory* is a 4-tuple $\mathcal{R}$=($\mathcal{F}$,E,$\mathcal{L}$,R) where $\mathcal{F}$ is a ranked alphabet of function symbols, E is the set of $\mathcal{T}(\mathcal{F},\mathcal{X})$-equalities, $\mathcal{L}$ is the set of labels and R a set of labelled rewrite rules of the form l : *lhs* $\rightarrow$ *rhs* where $l \in \mathcal{L}$ and *lhs, rhs* $\in \mathcal{T}(\mathcal{F},\mathcal{X})$ satisfying *Var(rhs)* $\subseteq$ *Var(lhs)*. The arity of l is the number of variables in *Var(lhs)*.

2.2 Computational Systems

The rewriting logic is proposed in [MOM96] as a logical framework in which other logics, like equational logic and Horn logic, can be represented and as a semantic framework for the specification of languages and systems. The operational semantics for rewrite theories are presented in [KKV95] where it is shown how computations in a rewrite theory can mirror computations in various logical systems.

The equivalence relation generated by E and the set of equational axioms (presented in [KKV95]) on the set of proof terms induces an equivalence on computations by considering that two computations are equivalent if their proof terms are equivalent.

A *strategy* is a subset of the set of proof terms $\mathcal{PT}$ of the current rewrite theory and is used to describe the computations one is interested in. The result of applying a strategy on a term is the set (possibly empty) of all terms that can be derived from the initial term using the strategy. We formally express this by giving a functional representation to the application of a strategy $\mathcal{S}$ on a term t:

$$\mathcal{S}(t) = \{[t']_E | \exists \pi \in \mathcal{S}, [t]_E \stackrel{\pi}{\Rightarrow} [t']_E\}$$

A *computational system* is defined as a labelled rewrite theory $\mathcal{R}$=($\mathcal{F}$,E,$\mathcal{L}$,R) together with a strategy $\mathcal{S}$. [KKV95] gives examples of languages and systems designed in this formalism.

2.3 The ELAN Environment

ELAN is a language for designing and executing computational systems. In ELAN, a logic can be expressed by specifying its syntax and its inference rules. The syntax can be described using mixfix operators and the inference rules are described by conditional rewrite rules. In order to guide the application of the rewrite rules

strategies are introduced. A full description of the language and its implementation is given in [BKK98] and a survey of several examples that have been developed using ELAN is presented in [BKK+96].

All rewrite rules are working on equivalence classes induced by the set of equations E that in the case of ELAN is restricted to associativity and commutativity axioms, for the symbols defined to be associative-commutative.

A labelled rewrite rule in ELAN is defined as a pair of terms built on functional symbols and local variables and additionally it can be applied under some conditions and it can use some local assignments. Several rules may have the same label, the resulting ambiguities being handled by the strategies. The rule label is optional. In this case it is the responsibility of the designer to provide a confluent and terminating set of unlabelled rewrite rules.

The application of the labelled rewrite rules is controlled by user-defined strategies while the no-named rules are applied according to a default normalisation strategy. The normalisation strategy consists in applying no-named rules at any position of a term until the normal form is reached, this strategy being applied after each reduction produced by a labelled rewrite rule.

The application of a rewrite rule in ELAN can yield several results due to the equational(associative-commutative) matching and to the **where** clauses that can return as well several results. Applying consecutively two rules is represented by the concatenation operator "**;**" and the non-determinism is handled by two basic strategy operators: **dont care choose**(**dc**) that returns at most one result un-deterministically chosen from the set of possible results of the application of the rule and **dont know choose**(**dk**) that returns all the possible results. The strategy **repeat*** apply sub-strategies in a loop until none of them is applicable.

The syntactic definitions and the rewrite rules can be implemented in ELAN in a modular way. Each module represents a computational system that can be parameterised and combined with other modules in order to build the whole system

3 The Theorem Prover

Using the framework presented in the previous section we map the rules of the predicate prover described by J.-R. Abrial [Abr96,Abr97] in a computational system whose computations describe proofs in the initial proof calculus.

We just briefly describe the format of inference rules and the way these rules are used, a more detailed presentation can be found in the B-Book [Abr96].

The conjectures to be proved are described by sequents consisting of a number of hypotheses **H** and a goal **P** and they are represented in the form: $\mathbf{H} \vdash \mathbf{P}$.

We use the notation presented in [Abr97] for describing the inference rules:

	Antecedents	Consequent
RuleName	$S_1, S_2, ...S_n$	S

where *RuleName* is the name that uniquely identifies the rule and S_i are the antecedents that can be sequents of the form $\mathbf{H} \vdash \mathbf{P}$ or side-conditions written

in English. Such a rule represents the fact that the proof of the consequent can be reduced to the proof of his antecedents. The rules that have no sequents in the antecedents represent axioms. A sequent can be proved by an axiom having it as consequent.

The theorem prover is described by a signature providing the syntax, an inference system defining the possible inferences and a search plan that selects at each step of the reduction the inference rules that can be applied. In our approach the inference rules are expressed using conditional rewrite rules and the search plan by a strategy that guides the application of the rewrite rules. Depending on the strategies used for applying the inference rules the reduction process can yield different proofs, i.e. different computations.

A sequent of the form $\mathbf{H} \vdash \mathbf{P}$ is proved to be valid when it is transformed using the inference rules in a set of valid sequents, this proof corresponding in ELAN in finding a derivation $\mathbf{(H \mid\!- P)} \stackrel{*}{\Rightarrow} \mathbf{\$}$ (where **$** represents the valid sequent) using the rewrite rules guided by the specified strategy.

In the development of the theorem prover we can distinguish 3 steps: the implementation of a propositional prover, its extention to a predicate prover and finally the introduction of the notion of equality. Each of the provers extends the previous one by adding a certain number of syntactic elements and inference rules. Since the strategies used in the three cases are quite similar we insist on the rules and strategies used in the propositional prover, a more detailed description of its extensions is presented in [CK97].

3.1 The Propositional Prover

We present now the syntax and the inference rules of the first level of the prover - the propositional prover. It is also described the strategy used to apply these rules. Each of the described rules will be presented in the form proposed by J.-R. Abrial in [Abr97] and by its implementation in ELAN.

The inference rules of the prover are applied on sequents having the form **Seq** ::= **Pred** |– **Pred** with **Pred** a predicate in the form:

Pred ::= **Pred** <=> **Pred** | **Pred** | => **Pred** | **Pred && Pred** |
Pred or Pred | ^**Pred** | **Prop**

The operator "**::**" is provided for describing rules with several antecedents thus, the syntax of **Seq** is extended with the clause **Seq** ::= **Seq :: Seq**. Intuitively, the rule describing the behaviour of this operator specifies that a sequent of this form is valid if all its components are valid.

We should point out that the logical operators **or**, **&&**, <=> are declared in ELAN as being associative-commutative so the matching concerning these operators is done modulo associativity- commutativity. The constants **TRUE** and **FALSE** identify the true and respectively the false predicate and **Hvide** an empty set of predicates. This symbol is used in order to represent sequents of the form $\vdash \mathbf{P}$ in our syntax, i.e. sequents with no hypotheses.

For each of the binary connectors (**AND, OR, EQV, IMP**) a collection of four inference rules is defined together with another two ones for **NOT** and

additionally a set of axioms that allow to stop the reduction process is provided. A detailed presentation of these rules can be found in [Abr97] and [Abr96].

We start by presenting a rule with two antecedents, the rules with only one antecedent are handled in a similar way.

AND4	$H \vdash Q$ $H \vdash P$	$H \vdash P \wedge Q$

For the case where the rule contains more than one antecedent the reduction strategy (*dedstrat*) is applied recursively on the antecedents and the combination of the two results is provided as final result. The strategy used is presented later on in this section. The representation of the rule **AND4** in ELAN is (with `H, P, Q : Pred; S1, S2 : Seq`):

```
[AND4] H |-  (P && Q)          =>         S1 :: S2
                        where S1 := (dedstrat) H |- Q
                        where S2 := (dedstrat) H |- P   end
```

In the current implementation the strategy *dedstrat* used in the local assignments is the strategy applied on the initial sequent but we can use some other strategy built by the user for proving the antecedents.

The other rules for the conjunction, disjunction, implication and equivalence are represented in the same way except one rule for the implication that transforms the sequent to be proved in a sequent with an additional hypothesis:

IMP4	H,P $\vdash$ Q	H $\vdash$ P $\Rightarrow$ Q

If we use a strategy where the **IMP4** rule is applied after none of the other rules is applicable then we can prove that only simple propositions can be added to the hypotheses, where by simple proposition we understand a proposition containing no binary logical connectors. In order to make this property explicit in our system we implement the **IMP4** rule by the following rules (with `Ps : Prop; H, Q : Pred`):

```
[IMP4a] H |-  (Ps |=> Q)       =>         H && Ps |- Q              end
[IMP4b] H |-  (^Ps |=> Q)      =>         H && ^Ps |- Q             end
```

By specifying that *Ps* is an object of **Prop** we prevent the rule to be applied on sequents of the form $\mathbf{H \vdash P \Rightarrow Q}$ when **P** is not a simple proposition. For example, if we consider the sequent $\mathbf{H \vdash (A \wedge B) \Rightarrow (A \wedge B)}$ a strategy trying to apply the **IMP4** rule at the beginning would give as result the sequent $\mathbf{H \wedge (A \wedge B) \vdash (A \wedge B)}$ that cannot be proved using the rules of the prover. Using the rules **IMP4a** and **IMP4b** we are sure that the hypotheses part consists of a conjunction of simple propositions independently of the strategy used.

Since in ELAN the no-named rules are efficiently implemented we can improve the performance of the prover by replacing the labelled rule that reflects that the predicates (`P && P`) and `P` are equivalent with a no-named rule that is used in the normalising strategy.

```
[] P && P    =>     P              end
```

The number of rules applied in order to prove a sequent depends as well on the strategy used to guide the application of these rules.

Two strategies are proposed for applying the rules presented above. In the first approach the strategy is described by a loop in which at each step we try to apply one of the rules presented above in the following order: one of the rules **AND, OR, IMP1, IMP2, IMP3, EQV, NOT**, one of the rules **AXM**, the rule **IMP4** and the auxiliary rule **TB**.

```
[] dedstrat  =>
     repeat*( dc( SetAON, SetAXM, SetIMP, SetTB ) ); dc(FR)      end
```

The rule **FR** is applied at the end of the strategy in order to check if the result is a valid sequent. If this is not the case we backtrack to the last choice point in the strategy *repeat* and we try the next applicable set of rules. If none of the rules from this level leads to a valid sequent we backtrack again and we continue like this until we obtain a proof for the input sequent or when all the possibilities have been tried without success.

For reasons of modularity we group together the similar rules by using a strategy **dont care choose (dc)** that tries to apply each of the enumerated rules in the specified order. The set **AON** below is an example of using this method:

```
[] SetAON  =>    dc(AND1,AND2,AND3,AND4,OR1,OR2,OR3,OR4,
          IMP1,IMP2,IMP3,EQV1,EQV2,EQV3,EQV4,NOT1,NOT2)         end
```

A second approach consists in trying to apply the strategy **SetAON** repeatedly and as soon as none of the rules used by the strategy is applicable we start to do the same thing for the next strategy **SetAXM**. We proceed in the same manner for the rest of strategies and after **SetTB** is no more applicable we reiterate the process from the beginning. The reduction process stops when none of the rules is applicable on the current sequent.

This strategy is represented in ELAN in the following way:

```
[] SetAON  => repeat*( dc(AND1,AND2,AND3,AND4,OR1,OR2,OR3,OR4,
               IMP1,IMP2,IMP3,EQV1,EQV2,EQV3,EQV4,NOT1,NOT2)) end
. . .
[] SetSEQ    =>  SetAON; SetAXM; SetIMP; SetTB                 end
[] dedstrat  =>  repeat*( dc(RuleSEQ) )                        end
```

The strategy *repeat*()* used to build the loop returns the same sequent as the one received as input if none of the rules tried inside it is applicable. Thus, we have to introduce a new rule **RuleSEQ** that verifies if anything new has been achieved by executing one time the loop (if the sequent received as input has been changed) and if not it returns the input without continuing the loop.

```
  [RuleSEQ] H |- B              =>    S
               where S := (SetSEQ) H |- B
               if neq_Seq(S, H |- B)            end
```

The difference between the two approaches is that while for the first one, after a rule is applied successfully we try to use rules starting with the ones with

high priority, the second strategy applies the rules from the same level of priority (rules grouped in a strategy Set*op_name*) until none of them is applicable and then continues with the rules from the next level.

The problem that can arise in the second case is that applying **IMP4a** or **IMP4b** sooner in the reduction process can lead to a longer proof path. We give a simple example that illustrates the behaviour of the two strategies. We use the sequent $\mathbf{Hvide} \vdash \mathbf{A} \Rightarrow (\mathbf{B} \Rightarrow \mathbf{A})$. The rules used for the first strategy are:

```
'IMP4a':        (Hvide && A) |- (C |=> A)
'AXM4':         $
```

and for the second one:

```
'IMP4a':        (Hvide && A) |- (C |=> A)
'IMP4a':        ((Hvide && A) && C) |- A
'AXM3':         $
```

The environment ELAN allows to get the trace of the computations executed and to obtain statistics about the application of the rewrite rules. We show this by providing the output obtained when we try to prove the sequent `(Hvide |- (A |=> (B |=> C)) |=> (A |=> B) |=> A |=> C`.

For the first strategy the following output is obtained using the ELAN interpretor:

```
[] start with term :
    Hvide|-((A|=>(B|=>C))|=>((A|=>B)|=>(A|=>C)))
[] result term:
   $
 Statistics:
 total time     (0.176+0.000)=0.176 sec (main+subprocesses)
 average speed  159 inf/sec
                0 nonamed rules applied,     0 tried
               28   named rules applied,   913 tried
 named rules
           applied   tried    rule for symbol
                 4      13    TB:Seq
                 9       9    FR:Seq
                 4      25    IMP3:Seq
                 2      19    IMP4a:Seq
                 4      17    IMP4b:Seq
                 2      21    AXM1:Seq
                 1      26    AXM5:Seq
                 2      28    AXM4b:Seq
```

The application of the second strategy on the same sequent yields the following result:

```
[] start with term :
    Hvide|-((A|=>(B|=>C))|=>((A|=>B)|=>(A|=>C)))
[] result term:
   $
```

```
Statistics:
total time     (0.360+0.000)=0.360 sec (main+subprocesses)
average speed  136 inf/sec
               0 nonamed rules applied,     0 tried
              49   named rules applied,   725 tried
named rules
          applied   tried    rule for symbol
                4      19    TB:Seq
                9       9    FR:Seq
                4      19    IMP3:Seq
                1      16    AXM4:Seq
                3      33    STOP:Seq
                5      24    IMP4a:Seq
                4      19    IMP4b:Seq
                1      19    AXM1:Seq
                2      18    AXM2:Seq
                1      16    AXM5:Seq
               15      24    RuleSEQab:Seq
```

We notice that the main difference between the two consists in the number of rules **IMP4a** applied, the rule **RuleSEQ** consisting practically in a test.

Another possibility for guiding the rewrite rules would be to use the implicit normalising strategy, in this case the rules have to be unlabelled. The advantage of this approach is the efficient implementation of the unlabelled rules in ELAN but a great disadvantage is the lack of control in applying the rewrite rules. The statistics for such a strategy on the same input sequent are:

```
[] start with term :
    Hvide|-((A|=>(B|=>C))|=>((A|=>B)|=>(A|=>C)))
[] result term:
   $
 Statistics:
 total time     (0.052+0.000)=0.052 sec (main+subprocesses)
 average speed  596 inf/sec
               31 nonamed rules applied,   381 tried
                0   named rules applied,     0 tried
```

This normalisation strategy is not appropriate for the predicate prover where multiple instantiations for universally quantified predicates should be obtained by strategies that returns all possible results.

We should point out that not only the way the strategy operators are applied on the deductions rules is important but the order of these rules in the guiding strategy can be crucial in some cases. For instance a strategy that would try to apply the axioms only when none of the other main rules is applicable:

```
[] dedstrat  =>
      repeat*( dc( SetAON, SetIMP, SetAXM, SetTB ) ); dc(FR)      end
```

yields the use of 42 rules for the first strategy and 44 for the second one.

We can obtain all the possible proofs for a sequent by replacing the **dont care choose**(**dc**) operator with a **dont know choose**(**dk**) one that generates

all the possible matchings (due especially to associative-commutative operators) of rewrite rule on the input sequent and applies the rules in all the possible orders. For example, the rule **AND4** applied on $\mathbf{H} \vdash \mathbf{A} \wedge \mathbf{B} \wedge \mathbf{C}$ matches **P** on **A** and **Q** on $\mathbf{B} \wedge \mathbf{C}$ or **P** on $\mathbf{B} \wedge \mathbf{C}$ and **Q** on **A** and so on.

3.2 The Predicate Prover

The propositional prover is simple but in the same time very restrictive since we cannot reason about quantified predicates.

The syntax of the prover is extended with quantified predicates and we add a set of inference rules that transform the structure of the sequents received as input until they are proved or they are in the form **(H $\vdash$ FALSE)** and a rule (called **INS**) that tries to instantiate the quantified variables from sequents of this form with proper values in order to discover contradictions in the hypotheses and thus, to prove the respective sequents. We just present the rule **INS** and the way it is applied, the other rules are used in a similar way as the ones of the propositional prover.

In the **INS** rule we are looking for instantiations that transform the quantified predicates in simple predicates (or at least in predicates with a smaller number of quantified variables) that can lead further on to contradictions.

INS	Determine the instantiations P_1,P_2,...,P_n H $\vdash$ $P_1 \Rightarrow (P_2 \Rightarrow ...(P_n \Rightarrow$ FALSE)...)	H $\vdash$ FALSE

The search of proper instantiations is done by matching the universally quantified predicates on the already instantiated predicates from the hypotheses. Since for the matching problem we use a combined matching algorithm that combines algorithms for empty, commutative and associative-commutative theories we can obtain more than one result, each of them corresponding to different solutions of the matching algorithm.

In order to explore all the solutions of the matching algorithm we have to use a **dont know choose** strategy that allows to backtrack to a new instantiation if the application of the strategy on the previous one has failed.

```
[] SetINS  =>  dk(INS)        end
```

The main strategy looks similar to the one for the propositional prover and is containing the sets of rules for dealing with quantified predicates.

```
[] dedstrat  =>
     repeat*( dc( SetALLXST, SetAON, SetAXM, SetIMP, SetINS, SetTB ) );
     dc(FR)                                                          end
```

This strategy works like described in section 3.1 only that for the rule **INS** all its results will be considered when backtracking at its level.

4 Conclusions

We have shown how computational systems can be used as a logical framework for representing the first-order predicate prover proposed by J.-R. Abrial. The inference rules describing the two logics are naturally represented by conditional rewrite rules, the mapping of an inference rule from the form proposed by J.-R. Abrial in a rewrite rule in the system ELAN being in most of the cases a syntactical transformation.

Due to the semantics of ELAN that is based on many-sorted rewriting logic and to the possibility to define associative-commutative operators, we have been able to represent rules in a fairly intuitive way. By assigning types to the variables involved in the rewrite rules we have implicitly specified conditions on the structure of the terms that can be transformed by each rule thus, taking advantage of the well known benefit of typing objects. The definition of some operators as being associative-commutative has permitted to eliminate inference rules simulating these properties in the approach proposed by J.-R. Abrial. Some additional no-named rules have been introduced in order to improve the efficiency of the prover.

The application of all labelled rewrite rules is guided by strategies. Two approaches for defining them have been proposed and the results yielded by the two strategies have been compared. The effects of changing the order of the application of the rules have been presented and the use of an implicit normalising strategy has been considered. Although the differences between the proofs obtained with the different strategies for a simple sequent are not significant, we have to point out that small changes in the guiding strategy can influence significantly the efficiency of the computational system in the case of more complicated conjectures.

One of the advantages of representing the prover in rewriting logic is that we are then able to use standard rewrite tools for proving properties of the system like the termination for the propositional prover ([CK97]).

Among the perspectives open by this work, one concerns the implementation of interfaces for integers as well as for sets. The current syntax of the prover can be easily extended to include new operators on integers and sets and new inference rules describing their properties have to be introduced. Proper strategies should be designed for guiding the application of these new rules.

Acknowledgements We sincerely thank Thierry Lecompte for helpful discussions on the Atelier B predicate prover and Peter Borovanský and Pierre-Etienne Moreau for their useful hints and suggestions concerning the implementation in ELAN .

References

[Abr96] J.-R. Abrial. *The B-Book.* Cambridge University Press, 1996. ISBN 0-521-49619-5.

[Abr97] J.-R. Abrial. Le prouveur de prédicat. Technical report, 1997.

[BKK+96] Peter Borovanský, Claude Kirchner, Hélène Kirchner, Pierre-Etienne Moreau, and Marian Vittek. ELAN: A logical framework based on computational systems. In José Meseguer, editor, *Proceedings of the first international workshop on rewriting logic*, volume 4 of *Electronic Notes in TCS*, Asilomar (California), September 1996.

[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.

[CK97] Horatiu Cirstea and Claude Kirchner. Theorem proving using computational systems: The case of the B predicate prover. In *Workshop CCL'97*, Schlob Dagstuhl, Germany, September 1997.

[DJ90] N. Dershowitz and J.-P. Jouannaud. *Handbook of Theoretical Computer Science*, volume B, chapter 6: Rewrite Systems, pages 244–320. 1990. Also as: Research report 478, LRI.

[JK91] J.-P. Jouannaud and Claude Kirchner. Solving equations in abstract algebras: a rule-based survey of unification. In Jean-Louis Lassez and G. Plotkin, editors, *Computational Logic. Essays in honor of Alan Robinson*, chapter 8, pages 257–321. Cambridge (MA, USA), 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. 1995.

[Mes89] J. Meseguer. General logics. In *Proc. Logic Colloquium '87*. North Holland, 1989.

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

[MOM96] Narciso Martí-Oliet and José Meseguer. Rewriting logic as a logical and semantic framework. In José Meseguer, editor, *Proceedings of the first international workshop on rewriting logic*, volume 4, Asilomar (California), September 1996. Electronic Notes in Theoretical Computer Science.