Multi-relational Data Mining, Using UML for ILP

Arno J. Knobbe[1,2], Arno Siebes[2], Hendrik Blockeel[3], and Daniël Van Der Wallen[4]

[1]Perot Systems Nederland B.V., Hoefseweg 1, 3821 AE Amersfoort, The Netherlands,
arno.knobbe@ps.net
[2]CWI, P.O. Box 94079, 1090 GB Amsterdam, The Netherlands
arno@cwi.nl
[3]K.U.Leuven, Dept. of Computer Science, Celestijnenlaan 200A, B-3001 Heverlee, Belgium
hendrik.blockeel@cs.kuleuven.ac.be
[4]Inpact B.V., Nieuwekade 201, 3511 RW Utrecht, The Netherlands
daniel@inpact.nl

Abstract. Although there is a growing need for multi-relational data mining solutions in KDD, the use of obvious candidates from the field of Inductive Logic Programming (ILP) has been limited. In our view this is mainly due to the variation in ILP engines, especially with respect to input specification, as well as the limited attention for relational database issues. In this paper we describe an approach which uses UML as the common specification language for a large range of ILP engines. Having such a common language will enable a wide range of users, including non-experts, to model problems and apply different engines without any extra effort. The process involves transformation of UML into a language called CDBL, that is then translated to a variety of input formats for different engines.

1 Introduction

A central problem in the specification of a multi-relational data mining problem is the definition of a model of the data. Such a model directly determines the type of patterns that will be considered, and thus the direction of the search. Such specifications are usually referred to as declarative or language bias in ILP [14]. Most current systems use logic-based formalisms to specify the language bias (e.g., Progol, S-CART, Claudien, ICL, Tilde, Warmr [13, 12, 6, 7, 3, 8]). Although most of these formalisms are quite similar and make use of the same concepts (e.g., types and modes), there are still differences between the formalisms that make the sharing of the language bias specification between engines a non-trivial task. The main reasons for this are:

- the different formalisms each have their own syntax; the user needs to be familiar with all of them
- many formalisms contain certain constructs, the semantics of which, sometimes in a subtle way, reflect behavioral characteristics of the inductive algorithm.

D.A. Zighed, J. Komorowski, and J. Żytkow (Eds.): PKDD 2000, LNAI 1910, pp. 1–12, 2000.
© Springer-Verlag Berlin Heidelberg 2000

The use of different ILP-systems would be simplified significantly if a common declarative bias language were available. Such a language should have the following characteristics:

- The common language should be usable for a large range of ILP systems, which means that it should be easy to translate a bias specification from the common language to the native language of the ILP system
- It should be easy to learn. This means it should make use of concepts most users are familiar with. In the ideal case, the whole language itself is a language that the intended users are familiar with already
- The bias should not just serve as a necessary prerequisite for running the induction algorithm, but should also be usable as a shared piece of information or documentation about a problem within a team of analysts with varying levels of technical expertise
- It should be easy to judge the complexity of a problem from a single glance at the declarative bias. A graphical representation would be desirable.

In this paper we propose the use of the Unified Modeling Language (UML) [2, 15, 16, 17] as the language of choice for specifying declarative bias of such nature. Over the past few years UML has proven itself as a versatile tool for modeling a large range of applications in various domains. For ILP the Class Diagrams with their usefulness in database modeling are specifically interesting. Our discussion will be based on these diagrams.

Why do we wish to use UML to express bias? First of all, as UML is an intuitive visual language, essentially consisting of annotated graphs, we can easily write down the declarative bias for a particular domain or judge the complexity of a given data model [9]. Another reason for using UML is its widespread use in database (as well as object oriented) modelling. UML has effectively become a standard with thorough support in many commercial tools. Some tools allow the reverse engineering of a data model from a given relational database, directly using the table specifications and foreign key relations. If we can come up with a process of using UML in ILP algorithms, we would then have practically automated the analysis process of a relational database. Finally, UML may serve as a common means of stating declarative bias languages used in the different ILP engines.

Although it is clear that UML is a good candidate for specifying first order declarative bias, it may not be directly clear how the different engines will actually be making use of the UML declarations. Its use in our previously published Multi-Relational Data Mining framework [10, 11] is straightforward, as this framework and the related engine Fiji2 have been designed around the use of UML from the outset. To translate UML bias declarations to logic-based bias languages, we use an intermediate textual representation, called Common Declarative Bias Language (CDBL). CDBL is essentially a set of Prolog predicates, which can be easily processed by the different translation procedures. Translation procedures for the popular engines Tilde, Warmr and Progol are currently available. The whole process of connecting an ILP engine to a relational database now becomes a series of translation steps as is illustrated by the diagram in figure 1. We have implemented and embedded each of these steps into a single GUI.

The investigation of UML as a common declarative bias language for non-experts was motivated by the efforts involved in the Esprit IV project Aladin. This project aims at bringing ILP capabilities to a wider, commercial audience by embedding a range of ILP algorithms into the commercial Data Mining tool, Clementine.

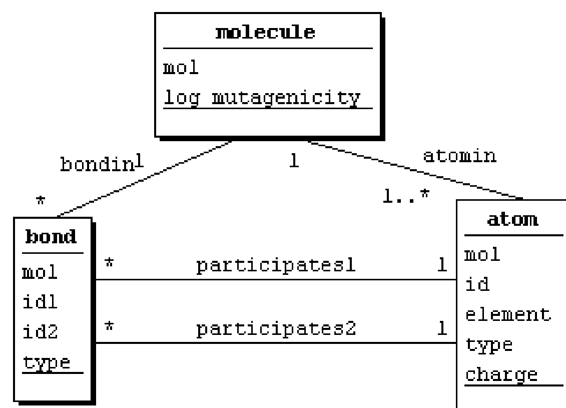

Fig. 1. The complete process of using UML with existing engines.

The outline of this paper is as follows. A section describing UML and its potential as first order declarative bias follows this introduction. We then give a short overview of the syntax of the Common Declarative Bias Language. In **Translating CDBL** we give an algorithm for translating CDBL to ILP input. Next we analyze the usefulness of UML as a declarative bias language compared to other approaches in **Comparing UML to traditional languages**. This section is followed by a **Conclusion**.

2 UML

From the large set of modelling tools provided by UML we will focus on the richest and most commonly used one: Class Diagrams [16]. These diagrams model exactly the concepts relevant for ILP, namely tables and the relation between them. In fact when we write UML in this paper we are referring specifically to Class Diagrams. There are two specific concepts within the Class Diagrams that we will be focusing on. The first is the concept of *class.* A class is a description of a set of objects that share the same features, relationships, and semantics. In a Class Diagram, a class is represented as a rectangle. Typically, a class represents some tangible entity in the problem domain, and maps to a table in the database.

The second concept is that of *association.* An association is a structural relationship that specifies that objects of one class are connected to objects of another. An important aspect of an association is its *multiplicity.* This specifies how many objects in one class are related to a single object in another, and vice versa. The multiplicity can thus be interpreted as the constraints on the association. Associations typically map to (foreign key) relations in the database although they sometimes need to be represented by an extra table, in the case of n-to-m relationships. An association

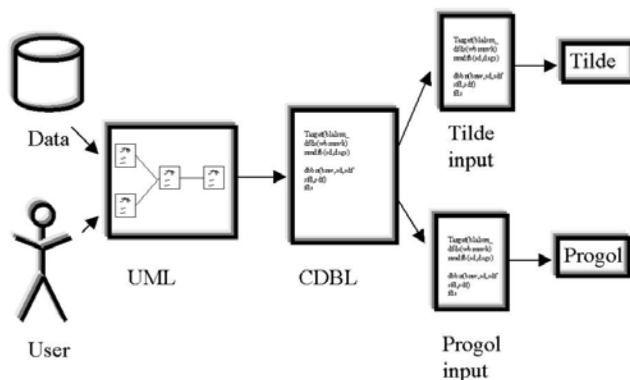

Fig. 2. UML model of the mutagenesis problem

is graphically represented as a line between two classes, which has certain labels (name, multiplicity, etc.) attached to it.

To illustrate these concepts, we show the Class Diagram describing part of the mutagenesis problem, a common benchmark problem within ILP [18]. As can be seen in figure 2, there are three classes: `molecule, atom` and `bond`. Four associations between these classes determine how objects in each class relate to objects in another class. As a bond involves 2 atoms (in different roles) there are two associations between `atom` and `bond`. The multiplicity of an association determines how many objects in one class correspond to a single object in another class. For example, a molecule has one or more atoms, but an atom belongs to exactly one molecule.

In order to see how UML may serve as input to an ILP engine we have to examine how the elements of a Class Diagram map to concepts in logic programming. An obvious way of doing this is to make classes and their attributes correspond to predicates and their arguments [8]. We do not specify whether these predicates need to be extensional or intentional. That is, both data originating from relational databases, as well as from Prolog databases including background knowledge can be modelled. Associations between predicates are used as constraints on the sharing of variables between these predicates, much in the same way as foreign links in [19]. Associations join two attributes in one or two predicates, which in logic programming terms means that a single variable may occur in the two positions in the argument lists identified by the two attributes.

Not only the existence of associations, but also the nature thereof in terms of its multiplicity provides bias. We will use this knowledge in three possible ways [10]:

- **Look-ahead**. For some refinements to a clause, the set of objects that support the clause is not actually changed. Therefore, it may be desirable to use a look-ahead in the form of extra refinements [3]. Given the multiplicity of an association involved in a refinement, we can directly decide whether this refinement changes anything to the support of the clause, and therefore whether the look-ahead is necessary. For example, in the mutagenesis domain a clause `molecule(…)` can be extended to form `molecule(X, M), atom(X, Y, N, O, P)`. However, this refinement provides the same support, as all molecules have at least one atom (multiplicity 1..*n*). A

look-ahead such as `molecule(X, M), atom(X, Y, carbon, O, P)` would make more sense.

- **Non-determinate**. Associations that have a multiplicity of *n* on at least one side are called non-determinate. Non-determinate associations indicate that a variable introduced in one literal may be bound in several alternative ways. Such multiple use of a variable has a large impact on the size of the search space. Testing large numbers of non-determinate clause can be avoided if the data model indicates that a particular association is determinate, i.e. has a multiplicity 1.
- **Mutual exclusion**. Some algorithms refine a single pattern into a set of derived patterns on the basis of the value of a single nominal attribute. The subsets belonging to these patterns do not necessarily form a partitioning of the original subset, in the sense that they may be overlapping. However, some algorithms, notably those for inducing decision trees, do require these subsets to be mutually exclusive. A multiplicity of 0..1 or 1 indicates that such a split may be made without the danger of overlap.

3 CDBL

In order to provide a textual format for the information stored in Class Diagrams that is suitable for interpretation by ILP engines, we introduce the Common Declarative Bias Language (CDBL). The aim of CDBL is to act as an intermediate language between UML and the specific declarative bias languages available for each ILP engine. As the translation from the graphical model to CDBL is trivial, we only need to produce engine specific translation procedures for CDBL in order to have an automatic procedure for using UML as declarative bias.

A CDBL definition is built up of a set of statements, ground facts to a Prolog interpreter. Each of these statements provides information about an element of the graphical model. Each table, association or constraint is thus described by a single statement.

One of the ideas behind the design of CDBL is the clear distinction between declarations about the relational bias, which is formed by restrictions from the data model, and search bias, which contains restrictions that are relevant to the current search strategy. The relational bias can usually be derived from the database schema automatically, whereas the search bias depends on the particular interest of the user. In most cases, one will want to try different forms of search bias, but keep the same relational bias.

Relational bias The relational bias is specified by expressions of the sort:

```
table(TableName, AttributeList).
types(TableName, TypeList).
association(AssociationName, Table1, Attribute1,
    Table2, Attribute2, MinMultiplicity1,
    MaxMultiplicity1, MinMultiplicity2,
    MaxMultiplicity2).
```

The types can be one of three predefined types, `numeric`, `nominal` and `binary` and determine which operators can be used in conditions involving the attribute. They are not used to determine how tables can be joined.

Search bias The following statements, among others, are part of the search bias:

```
target(Table, Attribute).
```

This statement indicates that the single table, called `Table`, represents the concept to be learned. The (optional) second argument determines that one of attributes in Table is used as the primary target for prediction. Appointing the target in the set of tables determines what type of object is analysed, and thus for example how support and frequency of patterns are computed. Note that positive and negative examples appear in the same table, and can be identified by the value of `Attribute`.

```
direction(Association, Direction).
```

This statement indicates that refinements may only be made in a certain direction along association `Association` rather than both ways.

```
fix(Table, AttributeList).
```

This statement indicates that if table `Table` is introduced in a pattern, all of the attributes provided in `AttributeList` are fixed by conditions on their values.

The CDBL representation of the example introduced in the previous section looks as follows:

```
table(molecule, [mol, log_mutagenicity]).
table(atom, [mol, id, element, type, charge]).
table(bond, [mol, id1, id2, type]).

types(molecule, [nominal, numeric]).
types(atom, [nominal, nominal, nominal, nominal, numeric]).
types(bond, [nominal, nominal, nominal, nominal]).

association(participates1, atom, id, bond, id1, 1, 1, 0, n).
association(participates2, atom, id, bond, id2, 1, 1, 0, n).
association(atomin, atom, mol, molecule, mol, 1, n, 1, 1).
association(bondin, bond, mol, molecule, mol, 0, n, 1, 1).

target(molecule, log_mutagenicity).

fix(bond, [type]).

direction(atomin, left).
direction(bondin, left).
```

4 Translating CDBL

This section describes an algorithm for translating CDBL specifications into language specifications for one of the currently supported engines, the ILP system Tilde [3, 4]. Our translation algorithm for converting CDBL into Tilde specifications consists of four steps:

1. generate **predict** and **tilde_mode** settings
2. generate types for tables
3. generate types and rmodes for standard operators (comparison etc.)
4. generate rmodes, constraints and look-ahead specifications for tables

Step 1

Step 1 is trivial: based on the **target** specification in the CDBL input, the corresponding **predict** setting is generated; based on the type of the argument to be predicted Tilde is set into classification or regression mode.

Step 2

Step 2 is implemented as follows:

```
2.1 for each table T:
         2.1.1 read its type specification types(T, L)
         2.1.2 change each type t in L into a unary
         term with functor t and an anonymous variable
         as argument
2.2 for each association between T1.A1 and T2.A2 :
         2.2.1 unify the types of the corresponding
         attributes
2.3 ground the type specifications by instantiating
each variable to a different constant
```

For the mutagenesis example given above this yield:

```
Types = { molecule(nominal(1), numeric(2),
atom(nominal(1),
         nominal(3), nominal(4), nominal(5),
         numeric(6)), bond(nominal(1), nominal(3),
         nominal(3), numeric(7)) }
```

With these type specifications it is ensured that the same variable can only occur in the place of arguments that have a chain of associations between them. At the same time, information on the original types is still available; this information will be used to decide, e.g., which variables can be compared using numerical comparison operators.

Step 3

Step 3 consists of generating rmodes, types and possibly look-ahead for a number of standard operators (=, <, …). Based on the **compare** setting in CDBL, one can allow comparisons between variables that a) have the same type; b) represent the same attribute of different instances; c) represent different attributes of the same instance.

For each allowed operator (the user can specify which operators are allowed), rmode and type specifications for comparisons of variables with the same type can be generated in a trivial manner; e.g.,

```
rmode(+X < +Y).                  % argument of <
predicate are input arguments
type(numeric(X) < numeric(X)). % only comparisons
between same type allowed
```

If we want to allow only comparisons between the same attribute of different instances, respectively different attributes of the same instance, this cannot be specified using only type and mode declarations. Tilde provides the possibility to specify constraints of the form **constraint(A,B)** where A is a literal (or conjunction of literals) that is considered for insertion in a certain node in the tree and B is a condition that has to be true in order for the insertion to be valid. For instance, the following code could be used for these constraints:

```
same_attribute(X,Y) :-
          occurs(Lit1), Lit1 =.. [F|Args1],
          strict_member(X, Args1), occurs(Lit2), Lit2
          \== Lit1, Lit2 =.. [F|Args2], strict_member(Y,
          Args2).
in_same_tuple(X,Y) :-
          occurs(Lit), Lit =.. [F|Args],
          strict_member(X, Args), strict_member(Y,
          Args).

constraint(X<Y, same_attribute(X,Y)).
constraint(X<Y, in_same_tuple(X,Y)).

strict_member(A, [B|C]) :- A == B.
strict_member(A, [B|C]) :- strict_member(A, C).
```

Step 4

Step 4 is the most complicated step. The way our implementation works is as follows: for each association, the algorithm looks from between which tables the association runs, and it adds rmodes for the "to" table (and constraints on these rmodes) in accordance with the CDBL specifications (e.g., which arguments are fixed). At this point the algorithm also inspects the properties of the association and based on these possibly adds look-ahead specifications. The algorithm is shown below.

Its results are 1) a set of rmodes; 2) a set of constraints; 3) a set of look-ahead specifications. The constraints can be seen as an array of constraints indexed on tables. For each table the constraint is initialised to *false* and whenever an association in the CDBL specifications indicates that the table can be added if certain conditions are fulfilled, the currently existing constraint on this table is weakened by extending it disjunctively with the new conditions. The "needs look-ahead" test is based on inspection of multiplicities, as indicated earlier. When adding an rmode or a look-ahead specification, it is implicitly checked whether the rmode or look-ahead already occurs (if it does, it is not added again).

The algorithm can be described as follows.

```
Global variables: rmodes: set of rmodes; initialised to
the empty set.
Constraints: array of constraints indexed on tables

For each association Assoc between an attribute A1 of
some table T1 and an attribute A2 of some table T2 that
goes in the right direction:
      add an rmode R for T2
          the argument corresponding to A2 has mode +
          the arguments occurring in a fix list for T2
          have mode #
          all other arguments have mode -
      constraints(T2) := constraints(T2) OR (the
                         variable occurring as A2 in T2
                         already occurs as A1 in a T1
                         literal)
      if T1 needs look-ahead then create look-ahead(T1,
      T2);
      if T2 needs look-ahead then
          add_rmodes+constraints+look-ahead for
             comparing attributes of T2 with constants
      else
          add_rmodes+constraints for comparing
             attributes of T2 with constants
Add_rmodes+constraints[+look-ahead] for comparing
attributes of T with constants:
For each allowed operator op :
      add rmode(+X op #X)
      add a constraint stating that X must already
      occur inside a T literal in the clause
      [add look-ahead allowing test to be added
      immediately after adding T]
```

The above translation algorithm was tested on a few test cases. The results of these experiments suggest that the bias specifications typically generated by the algorithm are significantly more complex than the ones humans would usually write. This additional complexity is in part attributable to the mismatch between relational

concepts and logical concepts, as is explained in the next section. It should be noted, though, that the size of the hypothesis space does not increase due to the more complex specifications and thus the complexity of the specification is not harmful to Tilde's performance.

Although the translator described above was designed with the Tilde system in mind, it is also usable for the Warmr ILP system, which uses the same bias specification language as Tilde except for some minor differences (e.g., no target argument is to be specified for the target table).

5 Comparing UML to Traditional Languages

If we want to compare the usefulness of our approach to existing means of bias specification we have to take different criteria into account. Because one of our goals is to widen the audience of ILP by improving its usability, some of these criteria will be somewhat subjective. We will be looking at the ease of use of UML in the context of ILP as well as the comprehensibility of Class Diagrams for non-experts. However we will start our comparison with a more objective aspect which involves the contrast between the actual bias (i.e. the hypothesis space) produced by UML and more traditional languages.

The main difference in actual bias between UML and traditional declarative bias languages is in how sharing of variables is controlled. In UML associations are used to explicitly allow two arguments to share a variable. In other languages this is less strict and any pair taken from a set of arguments with the same type can share a variable. What effect this difference in restriction has, is best demonstrated by considering the simplest case of three classes, `a, b` and `c`, connected by two associations both referring to the same argument in the middle class `b`. Bias on the basis of types would allow both `a(X), b(X), c(X)` as well as `a(X), c(X)`, whereas UML would only consider the former.

Assume either of the two associations is compulsory with respect to `b` (multiplicity 1 or 1..*n* on the side of `b`). In this case the two expressions are logically equivalent and the bias is effectively the same. The use of UML has a slight preference, because only one of the two alternatives is considered, in fact the one closer to our intuition.

If, on the other hand, both associations are optional (0..1 or 0..*n*), then the two expressions are distinct and UML will only allow `a(X), b(X), c(X)`. If patterns such as `a(X), c(X)` should be considered we would have to explicitly introduce a third association between `a` and `c`.

From the discussion above we can conclude that the use of associations is more explicit than that of types. Associations can express all bias that can be expressed by types, because each sharing of variables of the same type can be listed explicitly. In the meantime more precise and restrictive forms of bias can be declared by stating only the required associations, something which is hard to achieve by types only.

Because UML is such a versatile and widely-used tool for system modeling it needs little arguing that the language is easy to use and comprehensible for technical roles. Comprehensibility will however be lower for people with a non-technical background. This view is supported by an empirical study at British Airways [1], which reported that comprehensibility of Class Diagrams is high for 'Analyst',

'Designer' and 'Programmer' roles. One of these, the 'Domain Expert', is moderate at understanding these diagrams through its exposure to OO-techniques when co-operating with Analysts. In short we can say that UML is a comprehensible tool for the intended audience, although some users may require a short introduction into the syntax.

Because of its unified nature UML is slightly limited in expressiveness for ILP purposes. The types of possible constraints are limited to relational and search bias, whereas dedicated input languages can offer a wider range of engine-specific constraints. Also, many declarative bias languages allow the expression of general constraints on refinements with the full power of Prolog. Although many of the desirable constraints can be derived from the relational bias, as was shown before, our method is clearly restricted in terms of expressing general constraints.

6 Conclusion

In this paper we have introduced a method that facilitates the use of ILP in a KDD setting. Our approach is centered around the use of UML as a declarative bias language for a wide range of existing ILP engines. UML has a number of advantages over existing languages that are specific to particular engines:

- easy to comprehend for a wide range of people
- close relation to relational database modeling
- industry standard for modeling since 1997
- generic specification language for range of ILP algorithms.

Of course the last advantage only holds with sufficient support from the different engines (currently three). However, UML clearly has inherent features, which make it a good candidate for a common declarative bias language.

Because UML can be seen as a common denominator of a range of bias specification languages, it can not be expected to cover every single construct in a particular language. In order to be able to use the full functionality of an engine the native language will still be necessary. The usage of UML will be attractive in those cases where functionality is required, which is shared by most engines. Typically our approach is preferred for business analysts with a clear understanding of the domain at hand, who have had some minimal training in (database) modeling and inductive techniques and intend to apply a range of tools for comparison.

References

1. Arlow, J., Emmerich, W., Quinn, J. *Literate Modelling – Capturing Business Knowledge with the UML,* In proceedings of UML '98, 1998
2. Blaha, M., Premerlani, W. *Object-Oriented Modeling and Design for Database Applications,* Prentice Hall, 1998
3. Blockeel, H., De Raedt, L. *Top-down induction of first-order logical decision trees*, Artificial Intelligence 101 (1-2), pages 285-297. 1998

4. Blockeel, H., Dehaspe, L. *Tilde and Warmr User Manual*, http://www.cs.kuleuven.ac.be/~ml/PS/TWuser.ps.gz. 1999
5. Blockeel, H., De Raedt, L. *Relational knowledge discovery in databases,* Proceedings of the Sixth International Workshop on Inductive Logic Programming, Volume 314 of Lecture Notes in Artificial Intelligence, 1996, Springer Verlag
6. De Raedt, L., Dehaspe, L. *Clausal Discovery,* Machine Learning 26, pages 99-146. 1997
7. De Raedt, L., Van Laer, W. *Inductive Constraint Logic*, In Proceedings of the 6th International Workshop on Algorithmic Learning Theory. Lecture Notes in Artificial Intelligence, vol. 997, pages 80-94. 1995
8. Dehaspe, L. *Frequent Pattern Discovery in First-Order Logic*, Ph.D. thesis, Katholieke Universiteit Leuven, 1998
9. Knobbe, A.J. *Towards Scalable Industrial Implementations of ILP,* ILPNet2 Seminar on ILP & KDD, Caminha, Portugal, 1998
10. Knobbe, A.J., Blockeel, H., Siebes, A., Van der Wallen, D.M.G. *Multi-Relational Data Mining,* In Proceedings of Benelearn '99, 1999
11. Knobbe, A.J., Siebes, A., Van der Wallen, D.M.G. *Multi-Relational Decision Tree Induction,* In Proceedings of PKDD '99, Prague, Czech Republic, September 1999
12. Kramer, S. *Structural regression trees*, In Proceedings of the 13th National Conference on Artificial Intelligence, pages 812-819. AAAI Press / The MIT Press. 1996
13. Muggleton, S. *Inverse entailment and Progol*, New Generation Computing 13, 1995
14. Nédellec, C., Rouveirol, C., Adé, H., Bergadano, F. and Tausend, B. *Declarative bias in ILP*, Advances in Inductive Logic Programming, pages 82-103. IOS Press, 1996
15. Object Management Group, 492 Old Connecticut Path, Framingham, Mass. *UML Notation Guide,* ad/97-08-05 edition, Nov 1997
16. Rumbaugh, J., Booch, G., Jacobson, I. *Unified Modeling Language Reference Manual,* Addison Wesley, 1998
17. Si Alhir, S. *UML in a Nutshell,* O'Reilly & Associates, Inc., 1998
18. Srinivasan, A., Muggleton, S., Sternberg, M.J.E. and King, R.D. *Theories for mutagenicity: a study in first-order and feature-based induction*, Artificial Intelligence 85, 1996
19. Wrobel, S. *An algorithm for multi-relational discovery of subgroups,* In Proceedings of Principles of Data Mining and Knowledge Discovery (PKDD '97), 78-87, 1997