ILP systems on the ILPnet systems repository

Irene Weber
Department of Computer Science, University of Stuttgart, Germany
email: Irene.Weber@informatik.uni-stuttgart.de

October 9, 1996

Contents

1 ILP systems on the web

This section gives an overview over the ILP systems gathered at the ILPNET systems repository. As this paper basically is a report on ILPNET, it emphasizes on systems developed by ILPNET members. For each system, a short description is given which summarizes its basic principles, the functionality and options it offers and the input it expects. To facilitate the selection of a suitable system, the hardware and software prerequisites for running it are described as well.

Descriptions of systems developed outside ILPNET are marked with an asterisk (*). These systems are described in less detail, expect for the system **FOIL** which is one of the best-known and successful empirical ILP systems and has inspired a lot of further research.

A common classification of ILP systems distinguishes between empirical ILP systems and interactive ILP systems [LD94, Rae92]. Empirical ILP systems are characterised as non-interactive batch learners inducing definitions for single predicates from scratch. Interactive systems, also called incremental systems, interactively learn multiple predicates, possibly starting with a preliminary incomplete or inconsistent theory. This classification is not to be taken too rigidly but rather as indicating two poles of the spectrum of exisiting ILP systems. To better fit the situation encountered at the ILP systems repository, we further divide the class of empirical systems according to the number of examples the systems expect to process. In the refined classification, the term *empirical ILP systems* then refers to single-predicate batch learning systems which are able to analyse large example sets requiring little or no user guidance.

Interactive ILP systems incrementally build complex domain theories consisting of multiple predicates where the user controls and initiates the subsequent steps of the model construction and refinement process. These systems usually offer a graphical user interface.

The third group of systems learns from small example sets in batch mode. One of the systems we assign to this group, namely **FILP** [BG93a, BG93b] queries the user for missing examples, but as this interaction takes place in advance to induction, and as the induction proceeds autonomously once the example set is completed, the learning algorithm in essence performs batch-learning. The system **MARKUS** [Gro92] is a non-interactive theory revisor learning single predicates, and consequently fits neither into the class of empirical ILP systems nor into the class of interactive ILP systems. Thus it seems natural to distinguish a third class of systems. As these systems require only small example sets and little user guidance, this class of systems may qualify as *programming assistants*.

Recently, the interest in ILP research has extended to include alternative task settings besides the classical ILP concept-learning task. The ILP systems repository includes two systems which are commonly assigned to the non-classical approach. They are described at the end of this section.

2 Empirical ILP systems

2.1 FOIL*

FOIL [QCJ93] is a system for learning intensional concept definitions from relational tuples. The induced concept definitions are represented as function-free Horn clauses, optionally containing negated body literals. The background knowledge predicates are represented extensionally as sets of ground tuples. **FOIL** employs a heuristic search strategy which prunes vast parts of the hypothesis space.

As its general search strategy, **FOIL** adopts a covering approach. Induction of a single clause starts with a clause with an empty body which is specialised by repeatedly adding a body literal to the clause built so far. As candidate body literals, **FOIL** considers the literals which are constructed by variabilising the known predicates, that is, by distributing variables to the argument places of background knowledge predicates. Additionally, **FOIL** takes into account literals stating (un)equality of variables. Furthermore, literals may contain constants which the user has declared as theory (i.e. relevant) constants.

All literals conform to the type restrictions of the predicates. For further control of the language bias, **FOIL** provides parameters limiting the total number and maximum depth of variables in a single clause. In addition, **FOIL** incorporates mechanisms for excluding literals which might lead to endless loops in recursive hypothesis clauses. **FOIL** offers limited number handling capabilities and generates literals comparing numeric variables to each other or to thresholds it has derived.

Among the candidate literals, **FOIL** selects one literal to be added to the body of the hypothesis clause. The choice is determined by the information gain heuristic. The gain heuristic is an information-based measure estimating the utility of a literal in dividing positive from negative examples. **FOIL** stops adding literals to the hypothesis clause if the clause reaches the predefined minimum accuracy or if the encoding length of the clause exceeds the number of bits needed for explicitly encoding the positive examples it covers. This second stopping criterion prevents the induction of overly long and specific clauses in noisy domains. Induction of further hypothesis clauses stops if all positive examples are covered or if the set of induced hypothesis clauses violates the encoding length restriction. In a postprocessing stage, **FOIL** removes unneccessary literals from induced clauses as well as redundant clauses from the concept definition.

FOIL's greedy search strategy makes it very efficient, but also prone to exclude the intended concept definitions from the search space. Some refinements of the hill-climbing search alleviate its short-sightedness, such as including a certain class of literals with zero information gain into the hypothesis clause and a simple backtracking mechanism.

FOIL is a batch learning system which reads in all learning input from a single input file. For learning, positive as well as negative examples are required. A user may provide negative examples explicitely or, alternatively, instruct FOIL to construct negative examples automatically according to the Closed World Assumption (CWA). In the latter case, the set of positive examples must be complete up to a certain example complexity. For predicates with high arity, the CWA may generate a huge number of negative examples. FOIL offers a command line option allowing the user to specify the percentage of randomly-selected negative examples to be used for induction.

Examples and background knowledge for FOIL have to be formatted as tuples, that is, each ground instance of a predicate is represented as a sequence of argument values. For each predicate, the user provides a header defining its name and argument types. Optionally, the user may indicate the input/output mode of the predicates, thus further limiting the number of literals constructed by FOIL.

For convenient testing of the induced hypothesis, the user may provide test cases (i.e. classified examples) for the target predicates together with the learning input. FOIL then checks the hypothesis on these cases and reports the results.

FOIL is available as C source code which is easily compiled using the enclosed Makefile. The release also includes a conversion program for transforming C4.5 input files into FOIL's format. The development of FOIL was started in 1989. Version 6.4 of FOIL, dating from January 1996, is available by `ftp://ftp.cs.su.oz.au/pub`.

2.2 MFOIL

The system MFOIL [LD94, Dže93] is a descendant of FOIL which aims at improving its noise handling capacities which are of crucial importance when processing imperfect real-world datasets.

MFOIL integrates several noise-handling techniques from attribute-value learning approaches into FOIL. It offers two alternative accuracy-based search heuristics replacing FOIL's entropy-based information gain criterion, namely the Laplace-estimate and the more sophisticated m-estimate. The m-estimate takes into account the prior probabilities of examples, leading to a more reliable criterion for small example sets. The user-settable parameter m allows to control the influence of the prior probabilities. In MFOIL, FOIL's encoding-length based stopping criteria are replaced by criteria relying on statistical significance testing.

Further differences between FOIL and MFOIL concern the search strategy and the background knowledge. As FOIL, MFOIL adopts a covering strategy, but, unlike FOIL, it conducts beam search in order to overcome at least partially some of the disadvantages of FOIL's greedy hill-climbing search. On the other hand, some of FOIL's more advanced features, such as number handling, are not realised in mFOIL. Whereas **FOIL** is restricted to ground background knowledge, mFOIL is able to process intensionally defined background predicates as well. Furthermore, compared to **FOIL**, mFOIL allows the user to declare additional informations on the background predicates which reduce the number of possible body literals constructed during induction and thus help to gain efficiency.

mFOIL is available as Prolog source code running with Quintus Prolog at `http://www.gmd.de/ml-archive/ILP/public/software/mfoil`. As reported in [LD94], this implementation of mFOIL runs considerably slower than **FOIL**.

2.3 GOLEM

As **FOIL**, **GOLEM** [MF92] is a "classic" among empirical ILP systems. It has been applied successfully on real-world problems such as protein structure prediction [KMLS92] and finite element mesh design [DM92].

GOLEM copes efficiently with large datasets. It achieves this efficiency because it avoids searching a large hypothesis space for consistent hypotheses like, for instance, **FOIL**, but rather constructs a unique clause covering a set of positive examples relative to the available background knowledge. The principle is based on the relative least general generalisations (rlggs) introduced by Plotkin [Plo71a, Plo71b]. **GOLEM** embeds the construction of rlggs in a covering approach. For the induction of a single clause, it randomly selects several pairs of positive examples and computes their rlggs. Among these rlggs, **GOLEM** chooses the one which covers the largest number of positive examples and is consistent with the negative examples. This clause is further generalised. **GOLEM** randomly selects a set of positive examples and constructs the rlggs of each of these examples and the clause obtained in the first construction step. Again, the rlgg with the greatest coverage is selected and generalised by the same process. The generalisation process is repeated as long as the coverage of the best clause stops increasing. **GOLEM** conducts a postprocessing step, which reduces induced clauses by removing irrelevant literals.

In the general case, the rlgg may contain infinitely many literals. Therefore, **GOLEM** imposes some restrictions on the background knowledge and hypothesis language which ensure that the length of rlggs grows at worst polynomially with the number of positive examples. The background knowledge of **GOLEM** is required to consist of ground facts. For the hypothesis language, the determinacy restriction applies, that is, for given values of the head variables of a clause, the values of the arguments of the body literals are determined uniquely. The complexity of **GOLEM**'s hypothesis language is further controlled by two parameters, i and j, which limit the number and depth of body variables in a hypothesis clause.

GOLEM learns Horn clauses with functors. It may be run as a batch learner or in interactive mode where the induction can be controlled manually.

GOLEM is able to learn from positive examples only. Negative examples are used for clause reduction in the postprocessing step, as well as input/output mode declarations for the predicates the user may optionally supply. For dealing with noisy data, **GOLEM** provides a system parameter enabling the user to define a maximum number of negative examples a hypothesis clause is allowed to cover.

GOLEM is coded in C. It is provided as an executable running on Sun SparcStations. Additionally, a tar-file is available containing the source files, a README file explaining the usage of **GOLEM** and some example datasets. A copy of **GOLEM** is found at
`http://www.gmd.de/ml-archive/ILP/public/software/golem`.

2.4 LINUS

LINUS [LD94] is an ILP learner which incorporates existing attribute-value learning systems. The idea is to transform a restricted class of ILP problems into propositional form and solve the transformed learning problem with an attribute-value learning algorithm. The propositional learning result is then re-transformed into the first-order language. On the one hand, this approach enhances the propositional learners with the use of background knowledge and the more expressive hypothesis language. On the other hand, it enables the application of successful propositional learners in a first-order framework. As various propositional learners can be integrated and accessed via **LINUS**, **LINUS** also qualifies as an ILP toolkit offering several learning algorithms with their specific strengths. The present distribution of **LINUS** provides interfaces to the attribute-value learners **ASSISTANT**, **NEWGEM**, and **CN2**. Other propositional learners may be added. **LINUS** can be run in two modes. Running in **CLASS** mode, it corresponds to an enhanced attribute-value learner. In **RELATION** mode, **LINUS** behaves as an ILP system. Here, we focus on the **RELATION** mode only.

The basic principle of the transformation from first-order into propositional form is that all body literals which may possibly appear in a hypothesis clause (in the first-order formalism) are determined, thereby taking into account variable types. Each of these body literals corresponds to a boolean attribute in the propositional formalism. For each given example, its argument values are substituted for the variables of the body literal. Since all variables in the body literals are required to occur also as head variables in a hypothesis clause, the substitution yields a ground fact. If it is a true fact, the corresponding propositional attribute value of the example is true, and false otherwise. The learning results generated by the propositional learning algorithms are retransformed in the obvious way. The induced hypotheses are compressed in a postprocessing step.

In order to enable the transformation into propositional logic and vice versa, some restrictions on the hypothesis language and background knowledge are necessary. As in most systems, training examples are ground facts. These may contain structured, but nonrecursive terms. Negative examples can be stated explicitly or generated by **LINUS** according to the CWA. **LINUS** offers several options for controlling the generation of negative examples.

The hypothesis language of **LINUS** is restricted to constrained deductive hierarchical database clauses, that is, to typed program clauses with nonrecursive predicate definitions and nonrecursive types where the body variables are a subset of the head variables. Besides utility functions and predicates, hypothesis clauses consist of literals unifying two variables (X = Y) and of literals assigning a constant to a variable (X = a). Certain types of literals may appear in negated form in the body of a hypothesis clause.

Background knowledge has the form of deductive database clauses, that is, possibly recursive program clauses with typed variables. The variable type definitions which are required to be nonrecursive have to be provided by the user. The background knowledge consists of two types of predicate definitions, namely utility functions and utility predicates. Utility functions are predicates which compute a unique output value for given input values. The user has to declare their input/output mode. When occuring in an induced clause, the output arguments are bound to constants. Utility predicates are boolean functions with input arguments only. For a given input, these predicates compute true or false.

LINUS is implemented in Quintus Prolog. The distribution provided at `http://www.gmd.de/ml-archive/ILP/public/software/linus` includes an executable of **CN2** [CB91] running with SunOS 4.1.3.

An empirical comparison of **FOIL**, **mFOIL** and **GOLEM** can be found in [Dže93]. [DL91] provides an empirical comparison of **LINUS** and **FOIL**.

2.5 PROGOL

The system **PROGOL** [Mug95] provides the user with a standard Prolog interpreter augmented with inductive capacities. **PROGOL** can be run interactively or in batch mode. In interactive mode, **PROGOL** behaves similar to a standard Prolog interpreter allowing the user to pose queries or assert new clauses. Additionally, the user can request the system to generalise the examples. In batch mode, **PROGOL** is called from the operating shell with the name of an input file containing examples and background knowledge as an argument.

PROGOL employs a covering approach like, e.g., **FOIL**. That is, it selects an example to be generalised and finds a consistent clause covering the example. All clauses made redundant by the found clause including all examples covered by the clause are removed from the theory. The example selection and generalisation cycle is repeated until all examples are covered. When constructing hypothesis clauses consistent with the examples, **PROGOL** conducts a general-to-specific search in the theta-subsumption lattice of a single clause hypothesis. In contrast to other general-to-specific searching systems, **PROGOL** computes the most specific clause covering the seed example and belonging to the hypothesis language. This most specific clause bounds the theta-subsumption lattice from below. On top, the lattice is bounded by the empty clause. The search strategy is an A^*-like algorithm guided by an approximate compression measure. Each invocation of the search returns a clause which is guaranteed to maximally compress the data, however, the set of all found hypotheses is not necessarily the most compressive set of clauses for the given example set. **PROGOL** can learn ranges and functions with numeric data (integer and floating point) by making use of the built-in predicates "is", <, =<, etc.

The hypothesis language of **PROGOL** is restricted by the means of mode declarations provided by the user. The mode declarations specify the atoms to be used as head literals or body literals in hypothesis clauses. For each atom, the mode declaration indicates the argument types, and whether an argument is to be instantiated with an input variable, an output variable, or a constant. Furthermore, the mode declaration bounds the number of alternative solutions for instantiating the atom. The types are defined in the background knowledge by unary predicates, or by Prolog built-in functions.

PROGOL's syntax for examples, background knowledge and hypotheses is Dec-10 Prolog with the usual augmentable set of prefix, postfix and infix operators. However, unlike in the Edinburgh DEC-10 Prolog syntax, a distinction is drawn in Progol between assertions, which are terminated in a "." and queries, which are terminated in a "?". Arbitrary Prolog programs are allowed as background knowledge. Besides the background theory provided by the user, standard primitive predicates are built into **PROGOL** and are available as background knowledge. Positive examples are represented as arbitrary definite clauses. Negative examples and integrity constraints are represented as headless Horn clauses. Using negation by failure (CWA), **PROGOL** is able to learn arbitrary integrity constraints. For instance, a clause $\boldsymbol{man}(X) \vee \boldsymbol{woman}(X) \leftarrow \boldsymbol{normal}(X)$, represented as the Prolog integrity constraint `:- normal(X), not(man(X)), not(woman(X)).` can be learned using the four mode declarations `:- modeh(1,false)?`, `:- modeb(1,normal(+p))?`, `:- modeb(1,not(man(+p)))?`, and `:- modeb(1,not(woman(+p)))?`, from examples like `:- normal(leslie1).`

PROGOL provides a range of parameters for controlling the generalisation process. These parameters specify the maximum cardinality of hypothesis clauses, a depth bound for the theorem prover, the maximum layers of new variables, and an upper bound on the nodes to be explored when searching for a consistent clause. **PROGOL** allows to relax consistency by setting an upper bound on the number of negatives that can be covered by an acceptable clause.

POGOL (Version 4.1) is distributed as C source code together with a manual and examples on `ftp://ftp.comlab.ox.ac.uk/pub/Packages/ILP/Progol4.1/`. It is freely available for academic research. **Progol** is also available under license for commercial research.

PROGOL4.2 is an upward compatible with version 4.1 but learns from positive-only data. The system is available from the author upon request. Papers describing **PROGOL**4.2, can be obtained as Postscript by anonymous `ftp` from `ftp.comlab.ox.ac.uk` in files `pub/Packages/ILP/Papers/poslearn1.ps` and `pub/Packages/ILP/Papers/slp.ps`.

2.6 SPECTRE

SPECTRE (SPECialization by TRansformation and Elimination) [BIA94] is an empirical ILP system that can handle large example sets very efficiently.

Given sets of positive and negative examples and an overly general initial theory, that is, a theory which covers all positive and some of the negative examples, **SPECTRE** specialises an overly general initial theory in order to find a hypothesis which entails all positive examples but no negative examples.

SPECTRE employs a divide-and-conquer technique to specialize the overly general hypothesis until no negative examples are covered. Specialisation of the theory is performed by combining clause removal with the transformation rule unfolding. When **SPECTRE** finds a clause that covers a negative example and no positive examples, it removes the clause. When it finds a clause that covers both negative and positive examples, it unfolds the clause. Unfolding a clause requires the selection of a literal of the clause. The clause is resolved with all clauses in the theory with heads unifying with the selected literal. Then, the clause is replaced by the resulting resolvents. The choice of which literal to unfold upon is made such that the entropy of the resolvents is minimized. The resulting hypothesis consists of those clauses that cover positive examples only.

SPECTRE uses the overly general theory as a declarative bias that not only restricts what predicate symbols may occur in bodies of learned clauses, but also how these can be invoked. Specialised clauses defining the target predicate can only contain literals which occur in the initial target predicate definition or in clauses resolving with the initial or intermediate target predicate definitions. Therefore, the initial theory is crucial for the success of the induction.

In order to run **SPECTRE**, two input files are needed, namely a theory file containing the overly general theory in the form of a Prolog program in the Edinburgh syntax, and an example file with unit clauses in the Edinburgh syntax defining the positive and negative examples as arguments of the predicates `pos`/1 and `neg`/1, respectively. **SPECTRE** assumes that the target predicate is non-recursive, i.e., the target predicate is not allowed to appear in bodies of clauses. Background predicates, however, may be recursive. **SPECTRE** provides a graphical interface. No system parameters need to be set.

SPECTRE (Version 1.0) is implemented in SICStus Prolog 3.1 and is provided both as a stand-alone application for SUN OS 4 and as a SICStus Prolog object file (requiring SICStus 3.1). **SPECTRE**'s homepage is located at `http://www.dsv.su.se/~henke/SPECTRE/SPECTRE.html`.

2.7 MERLIN

MERLIN (Model Extraction by Regular Language INference) [Bos96] is a non-interactive, multiple predicate learning system that has the ability to invent new predicates. Like **SPECTRE**, it uses an overly general hypothesis in the form of a logic program together with sets of positive and negative examples in order to find an inductive hypothesis which entails all positive examples but no negative examples.

The basic idea of the approach is to learn finite-state automata that represent allowed sequences of resolution steps. **MERLIN** first finds SLD-refutations for all examples using the overly general hypothesis, and then tries to find the minimal finite-state automaton that can generate all sequences of input clauses in the SLD-refutations of the positive examples and no sequences of input clauses in the SLD-refutations of the negative examples. After having learned the automaton, **MERLIN** produces a new theory that allows only those sequences of resolution steps which are allowed by both the initial theory and the learned automaton. This is done by calculating the intersection of the automaton and a grammar that corresponds to the overly general hypothesis. The intersection is used to derive the final hypothesis in the form of a logic program. During this process **MERLIN** may introduce new predicate symbols, i.e., it carries out a form of predicate invention.

As the induced hypothesis allows only such resolution sequences which are possible in the initial theory, the initial theory defines **MERLIN**'s language bias. For instance, as the resolution sequences produced by the new theory involve only those clauses which are also part of resolution sequences allowed by the initial theory (besides clauses defining the newly introduced predicates), it is important that all relevant clauses occur in the resolution sequences produced by the initial theory.

MERLIN expects as input a theory file containing the initial theory in form of a Prolog program in Edinburgh syntax, and an example file with unit clauses that define the positive and negative examples as arguments of the predicates pos/1 and neg/1, respectively. The arguments of the predicates pos/1 and neg/1 are instances of the same atom. **MERLIN** assumes that each positive example has at least one SLD-refutation such that there is no negative example with an SLD-refutation that has the same sequence of input clauses.

MERLIN provides a graphical interface. No parameters are to be set.

MERLIN (Version 1.0) is implemented in SICStus Prolog 3.1 and is provided both as a stand-alone application for SUN OS 4 and as a SICStus Prolog object file requiring SICStus 3.1. Merlin can be found at `http://www.dsv.su.se/~henke/MERLIN/MERLIN.html`.

2.8 FOIDL*

FOIDL [MC95] is a descendant of **FOIL** differing from its predecessor in the following three ways. First, **FOIDL** is able to process intensionally defined background knowledge. Second, it substitutes the assumption of output completeness for explicit negative examples. The output completeness assumption requires that a mode declaration for the target predicate is given. It states that for every unique input pattern appearing in the training set, all correct output patterns occur in the examples in training the set. Together with the mode declaration, the positive examples then implicitly determine the negative examples. The third difference between **FOIDL** and **FOIL** is that **FOIDL** supports the induction of decision lists. A decision list is an ordered set of clauses each ending with a cut. When answering a query, the decision list returns the answer of the first clause in the ordered set which succeeds in answering the query. **FOIDL** generates the clauses in the decision list in reverse order, that is, clauses learned first appear at the end of the decision list. As the covering algorithm tends to learn more general clauses covering many positive examples first the more general clauses are placed as default cases at the end of the decision list.

FOIDL (version 1.0 Alpha) is provided as source code for Quintus Prolog (version 3.1.3). This version of **FOIDL** only creates non-recursive programs and always creates decision lists, so it is useful only for functional, non-recursive concepts. Furthermore, two Lisp implementations of **FOIDL** are available, one version incorporating a Prolog interpreter, the other incorporating a Prolog compiler. The files are located at `ftp://ftp.cs.utexas.edu/pub/mooney/foidl/`.

2.9 FOCL*

The system **FOCL** [PK92] learns Horn clause programs from examples and, optionally, background knowledge. It integrates an explanation-based learning component with the inductive learning approach of **FOIL**. **FOCL** is able to use intensionally defined background knowledge and accepts as input a partial, possibly incorrect rule as an approximation of the target predicate. User-defined constraints which realise a declarative language bias allow to restrict the search space.

FOCL is available as a machine learning program in the form of Common Lisp source code running on a variety of machines, and as a Macintosh application.

The Macintosh version of **FOCL** adds a graphical interface to the machine learning program that graphs the search space explored by **FOCL**, so it is a useful pedagogical tool for explaining inductive and explanation-based learning. In addition, it provides facilities for creating and graphically editing knowledge-bases, tracing rules, and generating explanations, so the Mac version may be used as an expert system shell. **FOCL**'s homepage is located at `/www.ics.uci.edu/AI/ML/FOCL.html`.

2.10 HYDRA*

The relational concept learner **HYDRA** [AP93] extends the machine learning program **FOCL** by adding likelihood ratios to the induced classification rules. **HYDRA** learns a concept description for each class. The concept descriptions compete to classify test examples using the likelihood ratios assigned to clauses of that concept description. This reduces the algorithm's susceptibility to noise.

HYDRA is implemented in Common Lisp. The source code is available at `http://www.ics.uci.edu/~mlearn/Hydra.html`.

2.11 FORTE*

FORTE [RM95] (First Order Revision of Theories from Examples) is a system for automatically revising function-free first-order Horn clause theories. **FORTE** integrates a collection of specialisation and generalisation operators and conducts an iterative hill-climbing search through the space of revision operations. The system includes the operators delete-antecedent and delete-rule, adopted from propositional theory revision, **FOIL**-like operators for adding antecedents and new rules, two generalisation operators based on inverse resolution, and an antecedent-adding operator termed 'relational pathfinding'. Each iteration of the search identifies all possibilities for applying the operators. The revision operation resulting in a maximum increase in theory accuracy is performed. The revision process continues as long as revisions produce an improvement in accuracy or a reduction in theory size.

FORTE is provided as Quintus Prolog source code. A copy can be retrieved by `ftp://ftp.cs.utexas.edu/pub/mooney/forte/`.

2.12 CHILLIN*

CHILLIN [ZMK94] is an ILP algorithm combining elements of top-down and bottom-up induction methods. **CHILLIN**'s input consists of sets of ground facts representing positive and negative examples, and a set of background predicates expressed as definite clauses. Examples may contain functors. Basically, **CHILLIN** tries to construct a small, simple theory covering the positive, but not the negative examples by repeatedly compacting its current version of the program. Compactness is measured as the syntactic size of the theory.

The algorithm starts with a most specific theory, namely the set of all positive examples. Then it generalises the current theory, aiming to find a generalisation which allows to remove a maximum number of clauses from the theory while all positive examples remain provable.

Similar to GOLEM's approach, the generalisation algorithm finds a random sampling of pairs of clauses in the current program. These pairs are generalised by constructing their least-general-generalisations under theta-subsumption. If a generalisation covers negative examples, it is specialised by adding antecedents using a FOIL-like algorithm. If the specialisation with background predicates is not sufficient for preventing negative examples from being covered, CHILLIN tries to invent new predicates for further specialisation of the clause. At each step, CHILLIN considers a number of possible generalisations and implements the one that best compresses the theory.

CHILLIN is able to learn recursive predicates. It avoids generating theories leading to endless recursion by imposing syntactic restrictions on recursive predicates. However, CHILLIN may learn recursive predicates covering negative examples.

In the actual implementation, the CHILLIN algorithm employs a modified search strategy in order to gain efficiency. As reported in [ZMK94], the system is able to handle induction problems with thousands of examples when running on a SparcStation 2. CHILLIN (version 1.0 Alpha) is made available as source code running with Quintus Prolog version 3.1.3 by
`ftp://ftp.cs.utexas.edu/pub/mooney/chillin/`.

3 Programming assistants

3.1 FILP

FILP [BG93a, BG93b] is an interactive system which learns functional logic programs. Functional means that for each sequence of input values for a predicate there is exactly one sequence of output values the predicate produces. This restriction applies to the induced predicates as well as to the predicates defined in the background knowledge. The functions are required to be total, i.e., for any input, an output must exist. The restriction to functional programs does not significantly affect the expressive power as any computable function can be represented by a functional logic program. The requirement that functions should be total is more restrictive. However, in practise some non-total functions can be learned as well (for instance, quicksort not using the append predicate quicksort(X,Acc,Y)) provided that appropriate examples and background knowledge are specified.

The restriction to functional programs is central to the approach. As an explicit restriction of the set of allowed hypotheses, it makes the learning task a lot easier, since it excludes a priori many clauses which otherwise must be generated and checked against the examples. Furthermore, it enables FILP to learn from positive examples only, since negative examples of the behaviour of induced predicates are implicitly given as the ones with the same input values but with different output values than the positive examples.

Unlike some other approaches, **FILP** does not require an example set which is complete up to a certain example complexity for the predicates to be learned, since, due to the functionality requirement, **FILP** can determine the examples needed for learning besides the given ones. For collecting the missing information, **FILP** queries the user. It presents examples with instantiated input values, and the user fills in the corresponding output values. This allows to start learning with a very limited number of initial examples, and more examples are added on request. This interactive way of example input is more convenient for the user than specifying the whole set of examples in advance, since **FILP** asks for all and only for the examples it really needs.

The background knowledge can be defined intensionally or extensionally. When using extensional background knowledge, **FILP** collects missing information on these background predicates as well. **FILP** is able to induce an intensional definition for such background predicates, thus realizing multiple predicate learning. **FILP** provably learns complete and consistent programs, that is, programs that cover all positive and no negative examples. Furthermore, if a complete and consistent program exists, **FILP** is guaranteed to find it. This is not the case for other approaches using extensionally defined background knowledge without example completion.

FILP works with flattened clauses, where functions are transformed into predicates. Besides the example set and background knowledge, the information **FILP** needs for learning includes a set of all the literals which may occur as body literals in the definition of the induced predicate. This literal set determines the hypothesis space for learning. As it affects the efficiency of learning and the number of literals **FILP** requests, this literal set should be specified carefully.

Furthermore, the user has to provide mode declarations for the induced predicate as well as for all background predicates. These mode declarations specify which arguments of a predicate are input arguments and which are output arguments. An aditional means for the user to influence induction is to declare 'forbidden clauses', that is clauses which must not be part of a solution even if satisfied by the positive examples.

FILP learns list manipulation programs such as member, quicksort or reverse, which are induced within a few seconds. For example, learning a functional variant of the member predicate member(Elem,List,yes/no) required four examples and about 12 seconds [BG93b].

FILP is written in C-prolog (interpreted). The C-prolog interpreter is provided together with the system and runs on a SUN SPARCstation 1 as well as on a SUN4/200 under SunOS, but has problems running under Solaris. The system is obtained at `http://www.gmd.de/ml-archive/ILP/public/software/filp`.

3.2 LILP

The system **LILP** [Mar95] encorporates an approach to concept learning from positive-only examples whose basic technique is borrowed from ***lambda***-calculus. It relies on a generality ordering between Horn clauses called λ-subsumption, which is stronger than θ-subsumption and weaker than generalized subsumption. λ-subsumption allows to compare clauses in a local sense, i.e., with respect to a partial interpretation of the background knowledge. Consequently, the locality of λ-subsumption allows to search for clauses which are correct with respect to a small subset of the set of atoms they generally cover.

Induction of a single clause starts with a seed example. Variabilizing one argument of the seed example and keeping the other arguments fixed results in a logical expression which can be viewed as a function of the variable, say X. This function maps instances of X on *true* if the resulting instantiation of the expression matches an example, and on *false* otherwise. The set of *true* instances forms the λ *calculus model* of the expression. The algorithm searches a set of body literals which defines the set of true instances of the variable. This is done for each head argument in turn, and the found body literals are combined to a clause body. If more than one clause for a seed example is found, the system keeps the best one according to the proof complexity criterion. This technique is embedded into a covering algorithm. **LILP** postprocesses the clause set in order to remove redundant clauses. It does not alert the user if is not able to induce a clause set covering all positive examples.

Like **FILP**, **LILP** does not assume a Closed World in the sense that the set of positive examples up to a given example complexity must be complete. Rather, complete λ-models are required. **LILP** learns from positive examples, but is able to utilize negative examples if available. By providing negative examples, the user can force **LILP** to allow singleton head variables or to induce the more specific predicate definition when several predicate definitions of varied generality fit the given data.

When using **LILP**, the user has to specify the name and arity of predicates to be used as background knowledge as well as constants occuring in the examples and not be variabilised in the predicate definition, thus defining the system's language bias. The hypothesis language is further adjusted by two system parameters, one specifying the maximum number of body literals determining a single head argument, the other defining the maximum number of determinate literals in a clause. As **FILP**, **LILP** works with flattened clauses where functions are defined by predicates. Structured terms may occur, but are treated as constants. Due to implementation reasons, lists must not occur in examples. However, at a deeper level they are allowed.

The system is so fast as to induce, e.g., a definition for quicksort from extensionally defined background knowledge in less than a second.

Lilp is supplied as Prolog source code written for Poplog Prolog with Dec10 library at `http://www.gmd.de/ml-archive/ILP/public/software/lilp`. Running **Lilp** with Quintus Prolog requires some minor adaptions such as adding dynamic statements and loading libraries.

3.3 MARKUS

MARKUS [Gro92, Gro93], a derivative of Shapiro's Model Inference System **MIS** [Sha83], is a system for inducing Prolog programs from positive and negative examples. Like **FILP** and **LILP**, **MARKUS** employs a covering strategy. Unlike these systems, **MARKUS** induces clauses with functions and is able to start induction with a preliminary incomplement or inconsistent definition of the target predicate which is then refined.

For inducing single clauses, **MARKUS** searches a refinement graph. The search starts with the most general clause which is specialised by applying the refinement operators taken over from **MIS**. In contrast to **MIS**, **MARKUS** generates an optimal refinement graph, i.e., a refinement graph without duplicate nodes, thus improving efficiency. The refinement graph is searched by iterative deepening search. When **MARKUS** encounters a clause covering at least one yet uncovered positive and no negative example, the clause is added to the predicate definition. Redundant clauses are removed from the predicate definition.

As **MARKUS** realizes an exhaustive search, restricting the hypothesis space is crucial for learning. Therefore, **MARKUS** offers elaborate mechanisms for explicit and implicit declaration of the language bias. First of all, the language bias which is defined implicitly by the refinement operator can be adjusted to the learning task by choosing the appropriate refinement operator, as **MARKUS** offers a refinement operator for learning DCG clauses and a general refinement operator for inducing logic programs. The application of the general refinement operator is guided by input/output mode declarations and type definitions for the arguments of the target and background predicates specified by the user. A set of parameters determines the form of clauses generated by the refinement operator. E.g., these parameters allow to define the maximum number of body literals, and the maximum structure depth of head arguments. For further tuning of the search space, the user can specify various concrete syntactic restrictions within individual type and background predicate definitions in a uniform manner. E.g., these restrictions make it possible to define argument symmetry or to prevent particular combinations of literals.

MARKUS is a non-interactive learner, learning from positive and negative examples which offers various options for adjusting the system bias. It uses logic with functors and is able to generate hypothesis clauses with negated body literals. The background knowledge is defined intensionally. The user has to provide mode declarations and type definitions for the arguments of target and background predicates.

With default parameter settings, **MARKUS** learns quicksort from 4 positive and 3 negative examples in 24.317 seconds on a Sun4/100.

MARKUS is available at
`http://www.gmd.de/ml-archive/ILP/public/software/markus/`. The system runs without modifications on Quintus Prolog (Vax, Sun), SICStus Prolog (Sun, HP) and Arity Prolog (IBM PC).

4 Interactive ILP systems

4.1 MOBAL

MOBAL [MWKE93] is a knowledge acquisition environment which assists the user in developing a model of an application domain in a first-order logical representation formalism. It implements the balanced cooperative modeling paradigm where knowledge acquisition is viewed as a cyclic and highly interactive modeling process. The system comes with a convenient graphical user interface providing means for manual input and inspection of a domain model, and access to a range of tools covering many of the substasks involved in knowledge acquisition, e.g., automated discovery of rules, knowledge revision, and theory restructuring. Additionally, **MOBAL** facilitates integration of external ILP tools, thus extending its own method pool as well as adding an interactive graphical interface to non-interactive ILP programs.

MOBAL's internal learning algorithm **RDT** solves the ILP task by inducing rules from positive and negative examples. The negative examples can be listed explicitly by the user or, optionally, can be defined implicitly via the Closed World Assumption. The logical dialect used by RDT is the function-free subset of Horn clause logic extended by negated literals. Negation is not treated as negation by failure as, for instance, in **FOIL**, but rather as proper negation, i.e., the negation of an atom is considered as true only when there exists a corresponding negated fact.

RDT requires extensionally defined background knowledge for learning. If intensional predicate definitions are entered into the system, **MOBAL**'s built-in inference engine efficiently computes the corresponding extensional predicate definitions up to a prespecified depth limit. RDT's hypothesis space is spanned by a set of rule models specified by the user. A rule model is a rule where predicate variables replace actual domain predicates. **RDT** searches the hypothesis space defined by the given rule models by instantiating the predicate variables with compatible domain predicates.

Compatibility of predicates is defined with respect to a sort taxonomy and predicate topology holding in the domain model. The sort taxonomy divides the arguments of predicates into classes. The predicate topology reflects the inferential structure of the domain, i.e., it states which predicates are useful for defining other predicates. Ensuring that argument sorts and predicates in an instantiated rule model are compatible excludes useless rules from the hypothesis space. User-defined parameters control when to accept a rule as a hypothesis clause. Among others, these parameters take into account the number of positive and negative instances of the rule.

Rule models, sort taxonomy, and predicate topology enable very fine-grained adjustment of **RDT**'s hypothesis space with explicit user control, however at the price that their specification can be quite demanding. Assistance in this task is offered by **MOBAL**'s tools **MAT** (Model Acquisition Tool), **PST** (Predicate Structuring Tool), and **SST** (Sort Taxonomy Tool), which automatically derive rule models, a predicate topology, and a sort taxonomy from the current domain model. These provide insight into the structure of the domain model evolved so far and serve as a starting point for the specification of the structure of the intended domain model.

Further important components of **MOBAL** are a knowledge revision tool (**KRT**), a concept learning tool (**CLT**), and a theory restructuring tool (**RRT**). **KRT** assists in correcting inconsistencies which may arise while gradually developing a knowledge base. **MOBAL**'s concept learning component **CLT** learns concept definitions from examples. Whereas rule induction algorithms such as **RDT** or **FOIL** usually produce rules stating sufficient conditions for a concept, **CLT** also searches for necessary conditions as well as for all other rules that use the concept. These tools interact in the following way. When the knowledge revision process indicates that more concepts are needed for the compact representation of the revised knowledge base, **KRT** calls **CLT** for the generation of such concepts. Thus, the system realizes predicate invention. For learning rules for the new concepts, **CLT** calls **RDT**.

MOBAL's theory restructuring tool (**RRT**) assesses the quality of the domain theory according to a set of formal and statistical criteria and helps to clean up the knowledge base.

In summary, **MOBAL** is a complex and sophisticated system, and fully exploiting its functionality requires some training. Usage and getting started are facilitated by a comprehensive userguide and an online tutorial.

MOBAL is provided as an executable that runs on Sun Sparcs (sun arch 4) only. **MOBAL**'s interface is based on Tcl/Tk. As **MOBAL**'s README states, the windowing system (Open Windows, Motif, etc.) seems irrelevant as long as it is based on X11. The current version of **MOBAL** is developped under SunOS 4.1.∗, but running it under Solaris has shown to be unproblematic as well. **MOBAL**'s homepage is `http://nathan.gmd.de/projects/ml/mobal/mobal.html`.

4.2 MILES

MILES [ST93] is an ILP test environment designed to facilitate experiments with ILP methods and algorithms. It is not a ready-to-use ILP system in the sense that a user provides examples and background knowledge and the system returns a consistent theory. Rather, **MILES** contains an extensive collection of ILP operators used by common ILP algorithms and systems without embedding these operators into a fixed control mechanism and facilitates the integration of new operators. Thus, **MILES** is useful for investigating and comparing the effects of the available and newly defined operators as well as in teaching ILP methods. **MILES** facilitates rapid prototyping of specific ILP systems by adding a control mechanism which guides the application of operators taken from **MILES**'s operator pool. A generic control procedure is provided.

MILES contains five types of operators, namely generalisation operators, specialisation operators, for generalising or specialising clauses or sets of clauses, reformulation operators, preprocessing operators, and evaluation operators.

The generalisation operators generalise single clauses or sets of clauses. **MILES** provides six variants of *least general generalisation* operators (e.g., the *rlgg* operator used by **GOLEM**), nine *inverse resolution* operators and five truncation operators. There are four specialisation operators for single clauses (including three of the **MIS** operators also applied by **MARKUS**). Specialisation of sets of clauses is performed by a minimal base revision operator similar to the **MOBAL**'s revision operator KRT. This operator and the fourth single clause specialisation operator specialise by inventing new predicates. The reformulation operators transform the knowledge base equivalently in order to facilitate the learning task. **MILES** provides a reduction operator for single clauses and the flattening/unflattening operators which transform a knowledge base into function-free logic and vice versa.

MILES' preprocessing operators serve for extracting implicit informations from the examples and for initializing hypothesis clauses. The first operator automatically determines argument types for all example predicates. The second operator determines a set of clause heads covering the positive examples based on the structure of the example arguments.

The evaluation operators defined in **MILES** assess the quality of the knowledge base according to different criteria. **MILES** contains predicates for checking whether the knowledge base is complete and consistent with respect to the examples and procedures for detecting culprit clauses, in case that it is not complete nor consistent. **MILES** provides an operator for evaluating the clauses in the knowledge base by determining the positive and negative examples they cover as well as the derivations of examples in which the clauses participate. These informations allow to compute commonly employed measures as, for instance, the information gain used by **FOIL**. Furthermore, **MILES** contains two operators for computing the compression of the knowledge base. These operators can be instantiated with six different encoding schemes.

The generic control procedure available in **MILES** takes twelve parameters. These have to be instantiated with predicates defining the initialization of the hypothesis, a stopping criterion for the induction process, a quality criterion for accepting hypothesis clauses, the selection of the appropriate refinement operator etc. Example instantiations of the generic control realizing ILP algorithms for special types of logic programs (regular unary logic programs, definite clause grammars, constrained programs), and a **FOIL**-like algorithm are included.

Comparing **MILES** to **MOBAL** we find out that, although the systems contain similar components (knowledge base access and maintenance procedures, an inference mechanism, knowledge induction and revision operators, mechanisms for deriving argument types, theory evaluation criteria) they serve quite different purposes. Whereas **MOBAL** provides an ILP toolbox to be used for applications, **MILES** is a toolbox for investigation of and experimentation with ILP methods. Since **MILES** serves for rapid prototyping of ILP algorithms, it may also be viewed as an ILP construction kit. As **MILES** is a very flexible system providing a large range of operators with various parameters and options, a user either requires good knowledge of ILP or the willingness to acquire it. **MILES** offers a graphical X interface which, however, does not include access to the generic control. Usage and function of the operators are described in detailed comments in the source code of **MILES**.

MILES is provided as source code running with Quintus Prolog v3.1.1 or later. You can find it at `http://www.gmd.de/ml-archive/ILP/public/software/miles`. For using **MILES**'s X interface, v3.1.1 is strictly required, later versions will fail.

4.3 CLINT

CLINT [Rae92, DRB92] is an interactive theory revisor which allows the user to incrementally build and revise a logical knowledge base. It is often described as opportunistic, i.e., learning self-initiatively whenever possible, and user-friendly, i.e., easy to use without knowledge of its internal mechanisms.

CLINT is interactive in the sense that it queries the user in order to collect missing information on the predicates to be learned. The queries posed by **CLINT** are moderately complex, being either membership questions (the user is asked whether a given ground fact is true or false in the intended knowledge base), or existential questions (the user is asked to enter a ground substitution for a given non-ground fact such that the instantiated fact is true in the intended knowledge base). This contrasts to interactiveness in **MOBAL**'s sense, where interactive means that the user and the system's components cooperate and, in particular, that the system relies on the user to determine which step should be taken next.

CLINT's prominent features include the integration of an integrity theory defining constraints satisfied by the knowledge base, an abductive component, postponing of examples, parameterized language series for declaring and shifting the system's language bias, and a set of special multi-valued logical frameworks to be chosen by the user.

CLINT learns from positive and negative examples and from integrity constraints. Its hypothesis language is a subset of range-restricted functor-free Horn logic. Building and revising complete knowledge bases, **CLINT** is able to learn multiple predicates, so that the examples may concern several predicates.

CLINT's basic algorithm is a loop where the user repeatedly enters a new example or integrity constraint and **CLINT** revises the current theory in order to make it consistent with the new input. First, we briefly sketch the way **CLINT** processes examples. If the user enters examples consistent with the current theory, it suffices to simply remember these examples, otherwise the theory has to be revised. If a newly entered negative example is covered by the theory, **CLINT** searches the clauses used for proving the negative examples for an incorrect clause and retracts it. The incorrect clause is identified with the help of user queries. Positive examples which become uncovered by removing the incorrect clause are input into **CLINT**'s basic loop and thus trigger further theory revision. If **CLINT** encounters an uncovered positive example, it calls its abductive or inductive procedure. The abductive procedure completes the knowledge base by learning new facts. It constructs incomplete proofs of the uncovered positive example which are to be completed by entering ground facts to the theory (without making the theory inconsistent).

If no such facts can be found, the inductive procedure for learning new rules is triggered. **CLINT**'s induction algorithm consists of two steps. In the first step, the so-called justifications for positive example are generated. A justification is a most specific Horn clause covering the positive example and not covering any negative examples. In the second step, the justifications are generalised by dropping literals. **CLINT** relies on negative examples and user queries to determine which literals can be dropped safely.

The justifications which can be constructed for a positive example depend on the hypothesis language the system employs. In **CLINT**, parameterized languages are used for declaring the language bias. The language parameters specify the number and depth of variables in hypothesis clauses. The languages are ordered into sequences with increasing generality. If the system detects that the current language is not sufficient for learning a consistent predicate definition, it automatically shifts its bias to the next general language in the series. **CLINT** offers four predefined series, but the user may customize additional series as well.

Postponing of examples becomes necessary when none of the available languages contains a consistent justification for a positive example. In this case, the knowledge base misses relevant information. **CLINT** then automatically postpones the uncovered positive example until more information is available.

CLINT can learn from integrity constraints as well. An integrity constraint is a first-order clause describing properties of the knowledge base to be built. **CLINT** treats integrity constraints as generalised examples. Roughly speaking, this is done as follows. If **CLINT** detects that an integrity constraint is violated by the current knowledge base, the violated literal in the constraint is located via user queries. A violated body literal (i.e., a body literal that is wrongly true when instantiated by the violated substitution of the constraint) corresponds to a negative example, a violated head literal (i.e., a literal that is wrongly false) corresponds to a positive example. Both cases are passed into **CLINT**'s main loop and processed accordingly.

Among a range of other user-adjustable parameters, **CLINT** enables the user to select a suitable logical framework. Besides the default setting, where negation is treated as negation by failure, there are three variants of multi-valued logic, providing the truth-values inconsistent, unknown or both of them in addition to the truth-values true and false.

CLINT runs on Apple Macintosh computers under system 7 with at least two megabytes of free memory. It offers a convenient window-based interface designed according to Apple's standards. **CLINT** is available at `http://www.gmd.de/ml-archive/ILP/public/software/clint`.

5 Alternative ILP tasks

5.1 Inductive data engineering: INDEX

INDEX [Fla93] is a system for inductive data engineering. Inductive data engineering denotes the interactive process of restructuring a knowledge base by means of induction.

The underlying idea is to analyse a given database in order to detect hidden regularities which can be used for restructuring the database, yielding a more compact and meaningful representation. This approach does not conform to the classical ILP setting where the aim is to induce a hypothesis which, when combined with the background theory, explains the given examples, thus completing the theory. Rather, the approach uncovers information which is, though hidden, already present in the data. As Flach states, the process is nonetheless inductive, since it derives general rules from specific data [Fla93]. The general setting, of which **INDEX** realizes a variant, is called the nonmonotonic setting of ILP [MDR94] or the confirmatory setting of ILP [Fla95].

INDEX expects as input an extensional relation, that is, a set of ground facts defining the relation. This relation is searched for funtional and multi-valued dependencies among the attributes, i.e., arguments of the relation. Let X and Y denote sets of attributes. A functional dependency $X \rightarrow Y$ states that whenever two tuples in the relation have identical values of attributes X, they also have identical values of attributes Y. A multi-valued dependency $X \rightarrow\rightarrow Y$ generalises a functional dependency to sets of values of the dependent attributes. This means that the values of the dependent attributes Y are not determined uniquely by the values of the attributes X, but rather, each of a fixed set of attribute value combinations of Y occurs in the relation for given values of X. **INDEX** searches the relation for functional and multivalued dependencies in a **MIS**-like manner [Sha83], thereby exploiting the generality ordering of dependencies.

In the second step, **INDEX** restructures the relation, based on the attribute dependencies it has found to be holding in or violated by the relation. In either case, restructuring means that the relation is split into smaller relations allowing to reconstruct the original relation. Since the restructuring of the database involves the construction of new relations, INDEX performs predicate invention.

If an attribute dependency $X \rightarrow Y$ or $X \rightarrow\rightarrow Y$ is found to hold in the relation, it induces a so-called horizontal decomposition. The dependent attributes Y are removed from the original relation, and stored in a separate relation together with the determining attributes X. This yields two new, smaller relations from which the original relation is reconstructed by a JOIN operation on the common attributes X. This type of split of a relation is called a horizontal decomposition since it is to reverted by the horizontal JOIN operation [Fla93]. If INDEX discovers a satisfied attribute dependency, it automatically performs the horizontal decomposition and constructs the clause which expresses the join reverting it. INDEX queries the user to enter meaningful names for the new relations resulting from the decomposition.

Attribute dependencies which are violated in the relation induce vertical decompositions of a relation. The relation is divided into smaller relations such that the attribute dependency holds in each of the partial relations. This type of decomposition operation is reverted by UNION operations. In general, an attribute dependency induces many alternative decompositions of a relation. The selection of meaningful decompositions is guided by heuristics, but still requires user interaction.

INDEX is an experimental system which is not yet able to process large datasets. It is available as Quintus Prolog source code at
`ftp://ftp.gmd.de/MachineLearning/ILP/public/software/index/`.

5.2 Clausal discovery: CLAUDIEN

The interactive system CLAUDIEN [DRD95] performs the task of clausal discovery, that is, it searches a given database for hidden regularities. Both the database and the regularities are represented as first-order clausal theories. As the system INDEX, CLAUDIEN belongs to nonmonotonic setting of ILP. Whereas INDEX utilises detected regularities for restructuring the database, CLAUDIEN regards the discovered regularities as an aim of themselves. As CLAUDIEN provides a powerful mechanism for specifying the type of regularities to be detected, CLAUDIEN can be applied for detecting various kinds of regularities in databases, such as integrity constraints in databases, functional dependencies and determinations, or properties of sequences.

The basic principle of CLAUDIEN's discovery algorithm is to subseqently generate the clauses contained in the hypothesis language and check them against the database. The clauses which are found to represent an actual regularity of the data are added to the hypothesis. The algorithm searches the hypothesis space from general to specific, thereby exploiting the subsumption relations among of clauses for pruning the search space. While running, **CLAUDIEN** successively enlarges the set of discovered regularities. The longer the algorithm runs, the more regularities may be found. As the search outputs a valid hypothesis whenever it is interrrupted, **CLAUDIEN** can be regarded as an anytime algorithm.

CLAUDIEN's background theory and examples (termed 'observations') are represented as conjunctions of first order Horn clauses. The hypothesis may consist of arbitrary clauses. **CLAUDIEN** incorporates a mechanism for the syntactical declaration of the hypothesis language called **DLab**. This mechanism allows the user to specify general clause templates for hypothesis clauses. Each template defines sets of clauses. **DLab** derives refinement operators from the clause templates which map the expansion of the template into clause sets on sequences of specialisation operations under theta-subsumptions. This enables enables pruning of the search.

CLAUDIEN provides a range of control parameters. Some of these allow further control of the hypothesis language. Another group concerns semantical aspects of the hypothesis as, e.g., the minimum accuracy and coverage of discovered clauses. Additionally, the user can choose among four search strategies. Optionally, the system can be requested to produce non-redundant hypotheses, that is, hypotheses not containing clauses which are logically entailed by the background knowledge or other discovered regularities. Furthermore, **CLAUDIEN** provides mechanisms for the convenient management of different configurations of discovery experiments.

CLAUDIEN (version 3.0) is implemented in BIM Prolog. It is available for academic purposes as a stand-alone system running without a Bim Prolog system. If a Bim Prolog compiler is available, **CLAUDIEN** is able to make use of it, thereby significantly improving its execution times. The URL for **CLAUDIEN** is `http://www.cs.kuleuven.ac.be/cwis/research/ai/Research/claudien-E.shtml`

The **DLab** mechanism is also available as a Prolog library for the use with concept learning and knowledge discovery approaches other than **CLAUDIEN**. It is obtained via anonymous ftp at
`ftp://ftp.cs.kuleuven.ac.be/pub/logic-prgm/ilp/dlab`.

References

[AP93] K. Ali and M. Pazzani. HYDRA: A noise-tolerant relational concept learning algorithm. In *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence*. Morgan Kaufmann, 1993.

[BG93a] F. Bergadano and D. Gunetti. Functional inductive logic programming with queries to the user. In *Machine Learning: ECML-93, Eu-*

ropean Conference on Machine Learning, Wien, Austria. Springer, 1993.

[BG93b] F. Bergadano and D. Gunetti. An interactive system to learn functional logic programs. In *Proc. of IJCAI-93*, pages 1044 – 1049, Chambéry, 1993.

[BIA94] H. Boström and P. Idestam-Almquist. Specialization of logic programs by pruning sld-trees. In *Proc. of the 4th International Workshop on Inductive Logic Programming*, volume 237 of *GMD-Studien*, pages 31–48. Gesellschaft für Mathematik und Datenverarbeitung mbH, 1994.

[Bos96] H. Boström. Theory-guided induction of logic programs by inference of regular languages. In *Proc. of the 13th International Conference on Machine Learning*, pages 46–53. Morgan Kaufmann, 1996.

[CB91] P. Clark and R. Boswell. Rule induction with CN2: Some recent improvements. In *Proc. Fifth European Working Session on Learning*, pages 151–163, Berlin, 1991. Springer.

[DL91] S. Džeroski and N. Lavrac. Learning relations from noisy examples: An empirical comparison of linus and foil. In *Proc. of the Eighth International Conference on Machine Learning, Evanston.* Morgan Kaufmann, 1991.

[DM92] B. Dolsak and S. Muggleton. The application of inductive logic programming to finite element mesh design. In S. Muggleton, editor, *Inductive Logic Programming.* Academic Press, 1992.

[DRB92] L. De Raedt and M. Bruynooghe. An overview of the interactive concept-learner and theory revisor Clint. In Stephen Muggleton, editor, *Inductive Logic Programming*, pages 163–188. Academic Press, 1992.

[DRD95] L. De Raedt and L. Dehaspe. Clausal discovery. Technical report, Katholieke Universteit Leuven, 1995.

[Dže93] S. Džeroski. Handling imperfect data in inductive logic programming. In *Proc. Fourth Scandinavian Conference on Artificial Intelligence*, pages 111–125, Amsterdam, 1993. IOS Press.

[Fla93] P. Flach. Predicate invention in inductive data engineering. In P. Brazdil, editor, *Proc. of the Sixth European Conference on Machine Learning*, Lecture Notes in Artficial Intelligence, pages 83–94. Springer–Verlag, 1993.

[Fla95] P. A. Flach. *Conjectures: an inquiry concerning the logic of induction.* PhD thesis, Katholieke Universiteit Brabant, 1995.

[Gro92] M. Grobelnik. MARKUS — an optimized model inference system. In *Proc. ECAI'92 Workshop on Logical Approaches to Machine Learning*, Vienna, Austria, 1992.

[Gro93] M. Grobelnik. Induction of prolog programs with MARKUS. In *Proc. LOPSTR'93 Workshop on Logic Program Synthesis and Transformation*, Louvain, Belgium, 1993.

[KMLS92] R.D. King, S. Muggleton, R. Lewis, and M.J.E. Sternberg. Drug design by machine learning: the use of Inductive Logic Programming to model the structure-activity relationships of trimethoprim analogues binding to dihydrofolate reductase. In *Proc. Natl. Acad. Sci*, number 89, pages 11322–11326, 1992.

[LD94] N. Lavrač and S. Džeroski. *Inductive Logic Programming. Techniques and Applications.* Ellis Horwood, 1994.

[Mar95] Z. Markov. A functional approach to ILP. In L. DeRaedt, editor, *Proc. 5th International Workshop on Inductive Logic Programming*, Scientific Report. Department of Computer Science, K.U. Leuven, 1995.

[MC95] R.J. Mooney and M.E. Califf. Induction of first-order decision lists: Results on learning the past tense of English verbs. *Journal of Artificial Intelligence Research* 3, pages 1–24", 1995.

[MDR94] S. Muggleton and L. De Raedt. Inductive Logic Programming: Theory and methods. *Journal of Logic Programming*, Special Issue on 10 Years of Logic Programming, 1994.

[MF92] S. Muggleton and C. Feng. Efficient induction of logic programs. In S. Muggleton, editor, *Inductive Logic Programming.* Academic Press, 1992.

[Mug95] S. Muggleton. Inverse entailment and progol. *New Generation Computing Journal*, 13:245–286, 1995.

[MWKE93] K. Morik, S. Wrobel, J. Kietz, and W. Emde. *Knowledge Aquisition and Machine Learning: Theory Methods and Applications.* Academic Press, 1993.

[PK92] M.J. Pazzani and D¿ Kibler. The utility of knowledge in inductive learning. *Machine Learning* 9/1, pages 57–94, 1992.

[Plo71a] G. Plotkin. *Automatic methods of inductive inference*. PhD thesis, University of Edingburgh, 1971.

[Plo71b] G. Plotkin. A further note on inductive generalisation. In B. Meltzer and D. Michie, editors, *Machine Intelligence*, volume 6, pages 101–124. Edinburgh University Press, 1971.

[QCJ93] J.R. Quinlan and R.M. Cameron-Jones. Foil: A midterm report. In *Machine Learning: ECML-93, European Conference on Machine Learning*. Springer, 1993.

[Rae92] L. De Raedt. *Interactive Theory Revision: an Inductive Logic Programming Approach*. Academic Press, 1992.

[RM95] Bradley L. Richards and Raymond J. Mooney. Refinement of first-order horn-clause domain theories. *Machine Learning*, 19(2):95–131, 1995.

[Sha83] E. Y. Shapiro. *Algorithmic Program Debugging*. MIT Press, 1983.

[ST93] I. Stahl and B. Tausend. MILES – a modular inductive logic programming experimentation system. Deliverable STU1.2 of ESPRIT BRA 6020: Inductive Logic Programming (ILP), 1993.

[ZMK94] J. M. Zelle, R. J. Mooney, and J. B. Konvisser. Combining top-down and bottom-up techniques in inductive logic programming. In *Proceedings of the Eleventh International Workshop on Machine Learning (ML-94)*, pages 343–351, 1994.