Balanced Cooperative Modeling

Katharina Morik
University of Dortmund
Dept. Computer Science VIII
P.O.Box 500 500, D-4600 Dortmund 50
telephone: +49/ 231 755 5101
e-mail:morik@kilo.informatik.uni-dortmund.de

Abstract

Machine learning techniques are often used for supporting a knowledge engineer in constructing a model of part of the world. Different learning algorithms contribute to different tasks within the modeling process. Integrating several learning algorithms into one system allows it to support several modeling tasks within the same framework. In this paper, we focus on the distribution of work between several learning algorithms on the one hand and the user on the other hand. The approach followed by the MOBAL system is that of **balanced cooperation**, i.e. each modeling task can be done by the user or by a learning tool of the system. The MOBAL system is described in detail. We discuss the principle of multi-functionality of one representation for the balanced use by learning algorithms and users.

Key words:
multistrategy learning, balanced cooperative modeling, MOBAL

1 Introduction

The overall task of knowledge acquisition as well as the one of machine learning has often been described as constructing a model of part of the world, on purpose. If a system is to support the person constructing a model, it must **accept new items** and integrate them into the knowledge base. It must present the state of the domain model and allow the user to **inspect** it. It must support **revisions** of all modeling decisions of the user.

At last, it must support the **refinement** of rules or rule sets because of additional knowledge and the introduction of **new features** or concepts [1].

The first two requirements are fulfilled by most of the knowledge acquisition environments. Revisions are frequently supported only by a text editor. Then, it is up to the user to check consistency and integrity of the revised domain model. The user is supported only in performing the addition of new items and the inspection of the domain model.

Machine learning algorithms are most often used for automating the construction of rules as additional items of the knowledge base. Recently, automatic refinement and automatic construction of new features or concepts (constructive induction) are also provided by some machine learning systems. Moreover, inspection can also make good use of machine learning. Hence, for all of the modeling tasks listed above, there exists a machine learning tool which automatizes at least parts of it.

A system that integrates several learning tools, each responsible for performing a different modeling subtask, is a multistrategy learning system (Michalski 91). Questions concerning the **cooperation of tools** are whether one tool can use the results of another, whether several tools can use the same knowledge items, and whether a tool can call another one. MOBAL is such a multistrategy learning system, where the learning tools cooperate by means of input-output data so as to solve the global modeling task. But, also a multistrategy learning system needs some information about the domain and the desired domain model given by the user. The **cooperation with the user** is necessary even for the most advanced learning system. This is not a disadvantage. On the contrary, the user should guide the learning and be in control of the modeling process. On one hand we appreciate machine learning to automate some tasks. On the other hand we still want the users to perform their tasks - supported by the system. The question is how to organize the cooperation of user and system tools such that both, system and user contribute to model-building. For MOBAL, a synergistic effect can be stated which is the result of both, the user and the learning tools contributing to the global modeling task.

[1] For details of the modeling process see (Morik 89), (Morik 91)

2 Cooperation

There are different ways to use machine learning algorithms for knowledge acquisition. They correspond to a different distribution of work between system and user. The work share has consequences for the knowledge representation.

2.1 Work share between system and user

We may distinguish the following three prototypical ways of distributing the work between system and user in modeling a domain:

1. the one-shot learning where the user prepares examples and background knowledge and then runs an algorithm on the data;
   (examples are ID3 (Quinlan 83), FOIL (Quinlan 91), or KLUSTER (Morik, Kietz 89),(Kietz,Morik 91))

2. the interactive learning where the user prepares examples and background knowledge and then interacts with a learning system;
   (examples are DISCIPLE (Kodratoff,Tecuci 89) or CLINT (De Raedt, 91))

3. balanced interaction of system and user where learning contributes to the preparation of background knowledge, to enhancing the domain knowledge, and to inspecting the (learned) knowledge;
   (an example is MOBAL (described in this paper))

These options of how to use machine learning correspond to different **tasks** handled by a learning system. A learning task can be described by a certain type of input and the produced output. An additional characteristic of the learning task is, whether the learning is performed incrementally (which can be the case in the second and third option). Of course, the same learning task can be applied to various domains. In the first option, the user calls a learning algorithm for one particular task. Most often, this task is to learn a set of rules from examples of complementary classes. The second two options have the learning system cover a broader range of tasks. Each learning task corresponds to a learning tool which solves it - regardless of whether implemented as separate modules, or in one module.

Whereas in the first two options the user is requested to give some particular **information**, in the third option the user can give any information and the system uses it. Of course, the information must be sensible. However, the user is free to enter, e.g., facts or rules or term sorts or predicate sorts. That is, the distribution of work between system and user is strictly prescribed in the first two options whereas it is flexible in the third one.

The **control** of the modeling process is in the users' hands in the first option. They call the learning tool. In the second option, the system is in control. The system prompts the user to give the needed information. In the third option, control is mixed. The users can call tools explicitly as in the first option. If they don't want that, the flow of control between the tools is organized by the system. The users are never prompted to input a (counter-) example or a declaration of background knowledge. However, by setting some parameters they can state that they want to be asked by the system at certain decision points.

In the first option, **revisions** of the learning results are performed by the user in an edit-and-compile cycle with no more support than a text editor can give. If new, negative examples are acquired from the application, a new example set must be constructed, consisting of the new and some already known examples. The learning algorithm then constructs new rules which probably are better than the ones learned before. In the second option, some revision of rules is performed by a learning tool because of negative examples. In the third option, learning techniques are used for refinement and the construction of new features or concepts. Moreover, revisions of all modeling decisions that have been made are supported by some knowledge editing tools.

The prototypical ways of using learning (1-3 above) illustrate the aspects of work share between system and user:

- which tasks are performed by the user, which tasks are performed by the system?
- which information is given by the user, which information is constructed or derived by the system?
- is the user, the system, or are both in control of the modeling process?
- which revisions are supported by the system and which revisions are automatically done by a learning tool?

If the user as well as the system can perform a task, construct knowledge items of a certain kind, run (learning) tools, and revise given knowledge, then we call such a system **balanced cooperative**. MOBAL is such a balanced cooperative system. It will be described in detail in the next chapter.

2.2 Multi-functionality

The use of the system has consequences for the knowledge representation. In the first way of using a tool, the representation can easily be tailored for the needs of the one algorithm. The representation of a multistrategy learner (option 2 and 3) has to be designed with respect to several, possibly conflicting needs, or the different representations of different tools have to be integrated. The MOBAL system is a multistrategy learner which integrates various tools using a *uniform* representation. The integration problem with respect to knowledge representation is then to develop a formalism which is powerful enough to suit all tools well and which is still tractable. In contrast, the MLT system integrates several learning systems, each with its own representation (see Morik et al. 91). The integration problem is then to integrate *given* representation formalisms.

Balanced cooperative modeling allows the user as well as the system to work on the evolving domain model. As a consequence, all knowledge sources (examples, background knowledge, declarations, rules) have to be represented such that the system as well as the user can easily input, modify, and inspect the knowledge. This constrains the representation to be designed. If revisions of all knowledge entities have to be processed and their consequences have to be maintained by the system, this constrains the design of a representation even further.

The bi-directional use of knowledge bases has been discussed in other fields of artificial intelligence. For instance, a grammar is supposed to be used by the parser as well as by the generator of natural language sentences. Some efforts have also been made to use the same knowledge for plan recognition as for plan generation. Analogously, we claim that the same knowledge should be of good use for the user building up a model as for the learning system enhancing the model and building parts of it.

3 Cooperation in MOBAL

All the knowledge needed for problem solving in a particular domain can be input by the user. In this case, all the information is given by the user who performs all modeling tasks and completely controls the modeling process. The user is supported by an inference engine and a human-computer interface (see below). However, the user does not need to input almost everything. For each knowledge item which the user might input there exists a corresponding learning tool which can acquire parts of that knowledge. The basic input which the system expects from the user are facts and rule models (see next section). Of course, a system cannot create a model without any given information! But, also to those basic items there exist corresponding capabilities of MOBAL, namely the inference engine (deriving facts) and the model acquisition tool (producing rule models). Between the extremes of modeling by the user alone and some automatic contribution to the modeling by the system, all variations of work share are possible. This flexibility also has a disadvantage which should not be hidden: new users of the system miss the strict guidance which is given by interactive systems. They have difficulties selecting among all the possible choices.

3.1 MOBAL's representation

The MOBAL system is an environment for building up, inspecting, and changing a knowledge base. Before we present the learning tools, we describe the items which constitute a domain model in MOBAL.

The knowledge items integrated by the inference engine of MOBAL (Emde, 91) are:

- **facts**, expressing, e.g., relations, properties, and concept membership of objects;
  `owner(luc, diane1)` and `not(owner(luc,mercedes))` are facts
- **rules**, expressing, e.g., relations between concepts, necessary and sufficient conditions of concepts, hierarchies of properties;
  `owner(X,Y)&involved(Z,Y) --> responsible(X,Z)` is a rule
- **sorts**, expressing a structure of all the objects (constant terms) of the domain model;
- **topology of predicates**, expressing the overall structure of the rules of the domain model;
- **rule models**, expressing the structure of the rules to be learned;

The items are represented in a restricted higher-order logic which was proven to be tractable (Wrobel, 87). The user need not know all about the meta-predicates and the meta-rules in which they appear. The user also does not need to know the internal representation format. The windows of the human-computer interface provide presentations, both graphical and as text, of the knowledge base which are understandable without knowing the internal data structures. The user beginning an application regularly starts with facts and rules which are easy to understand. In the following, the knowledge items are described.

3.1.1 Facts

Facts are used to state relations, properties of objects, concept membership. Facts are represented as function-free literals without variables. The arguments of a predicate are of a particular *sort*. A fact `p(o1,o2,o3)` is only well-formed if the constant terms `o1, o2, o3` belong to the sorts of the first, second, or third argument place of `p`, respectively. For instance, the term at the first place of the predicate `involved` must be a member of the sort of events, the one at the second place must be a member of the vehicle sort. The form of a fact is $p(t_1, ..., t_n)$ where p is a n-ary predicate, t_j is a constant term or a number of the sort s_j.

The mapping from a fact to a truth value may obey a fuzzy logic because, in principle, the inference engine handles continuous truth values (Emde, 91). But, usually, it is difficult for a user to assign a fuzzy truth value to a fact. Therefore, only the truth values `unknown, true, false, contradictory` are used. A derived or input fact without explicit negation is interpreted as `true`. Every fact which is to be interpreted as `false` must be explicitly negated. This explicit negation has some advantages compared with the closed world assumption. It enables the user to input incomplete examples, to build up the model incrementally. The closed world assumption requires the user to know in advance which statements are necessary to complete the description of an example. But, as was stated above, modeling does not start with such a precise idea. Therefore, leaving out some statements in one example does not mean the negation of these statements. Hence, MOBAL interprets missing information simply as `unknown`.

Explicit negation also allows to explicitly contradict a derived fact of the system. Supposed, the inference engine has derived the fact `owner(luc, diane1)`

and the user knows that this is not true. The user then inputs

`not(owner(luc, diane1))`.

As a result the fact `owner(luc, diane1)`

becomes `contradictory`.

An explicit contradiction does not lead to the counter-intuitive behavior of standard logic that all formulas become true. Instead, the contradictory parts of the knowledge-base are excluded from inference processes. Hence, facts that are not contradictory keep their truth values. Contradictions are resolved by a knowledge revision component (Wrobel, 89).

3.1.2 Rules

In MOBAL, rules correspond to Horn clauses. In addition, the applicability of rules is maintained. For each variable occurring in a rule, its domain is represented as a *support set* (Emde, Habel, Rollinger 83; Wrobel, 89). In the normal case, the support set is a tuple of the sets of all objects. The rule `owner(X,Y) & involved(Z,Y) --> responsible(X,Z)` has a support set giving the domains for `X,Y,Z`. In the regular case, these are `all`. The support set then is `all x all x all`. But it is also possible to restrict the applicability of a rule to a more special support set. This can be done by exceptions of a variable's domain, by a tuple of exceptions of the support set, or by expressing a variable's domain by a concept. The above rule is only valid for events which are members of the concept `minor_violation`. The domain of variable `Z` is restricted to instances of minor (traffic law) violations: `all x all x minor_violation` is the correct support set for that rule. [2]

More formally, let `all` denote the set of all objects of a universe of discourse, D_i denote `all` or subsets of this set, T_j be a n-tuple of constant terms, and $t_1 ... t_k$ be constant terms (corresponding to particular objects in D), covered by a concept C, then the form of a support set for a rule with the variables $X_1, ..., X_n$ is:

$(X_1, ..., X_n)$ `in` $D_1 \times ... \times D_n$ `except` $\{T_1, ..., T_j\}$ where the `except` part can be empty.

The T_j are tuples of objects which should not be instances of the variables $X_1, ..., X_n$ of the rule because the rule would then lead to a contradiction.

[2] In Germany, the owner of a car has to pay a fine for a minor violation, even if he was not driving the car.

In a tuple T_j, each term can be of a different subset of `all`. In our example, such a tuple is `(luc,renault2,event3)`.

A particular D_i can be restricted by a set of exceptions, $\{t_1, ..., t_k\}$, written D_i `except` $\{t_1, ..., t_k\}$, in our example `all except` {`event3, event12, event13` }. Or, the variable's domain D_i is restricted to a particular concept as `minor_violation` in the example above.

3.1.3 Sorts

Sorts are used to guarantee the (semantic) well-formedness of predicates in facts and rules. Sorts can be named and given by the user in a predicate declaration: `owner/2: <person>, <vehicle>`. This means that the two-place predicate `owner` accepts only terms of sort `person` as the first argument and terms of sort `vehicle` as the second argument. It is not well-formed to state `owner(john, michael)`. A sort covers a subset D_i of `all`. The sorts which are built automatically by the sort taxonomy tool have constructed names, such as `arg1(owner)` denoting the set of terms occurring at the first place of the predicate `owner`.

Sorts with the same set of terms form a class. For instance, `arg1(owner)` and `arg1(responsible)` have the same constant terms, e.g. `[luc, yves, cathy]`. So, together they form a class:

`class21: [arg1(owner), arg1(responsible)] [luc,yves,eve]`

Classes are organized in a lattice. The most general class is `all`, the most special class is the empty class. There are subclasses and intersection classes. The lattice of classes gives an overview of all sorts and classes, their subset relations, their intersections. In this way, the structure of an application domain can be presented with respect to the objects of that domain. Figure 1 shows an excerpt of the lattice of sorts for the traffic law domain.

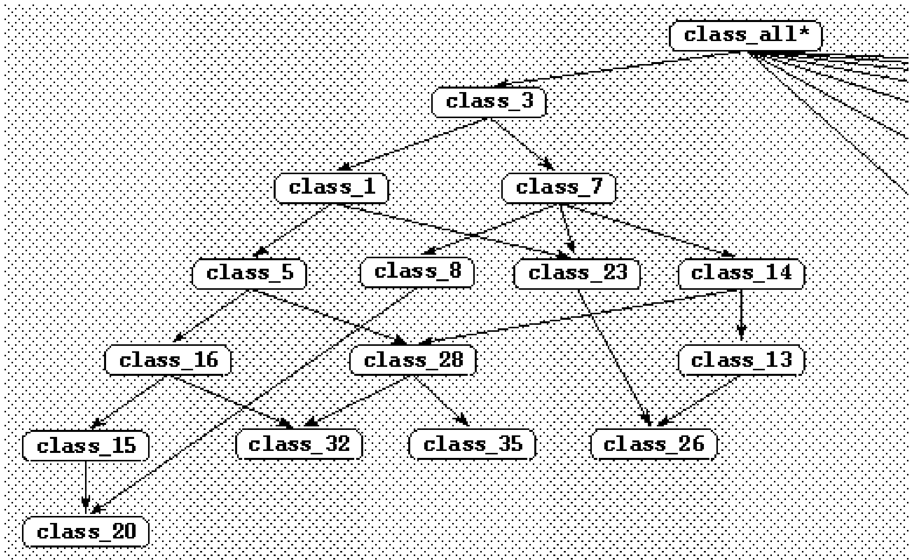

Figure 1: Excerpt of a lattice of sort classes

3.1.4 Topology of Predicates

The topology of predicates is used to guarantee the (semantic) well-formedness of rules. Sets of predicates form a named node of a graph. For instance, the node called `Beurteilung` (english: evaluation) represented as

`tnode: Beurteilung -Preds: [illegal_parking, responsible, unsafe_vehicle_violation]`
`-Links: [places,circumstances,laws]`

contains the predicates `illegal_parking, responsible, unsafe_vehicle_violation`. In the graph, the subnodes of this node are called

`[Orte, Umstaende, Verbote/Gebote, Fahrzeug, Verhalten]`

(english: [places,circumstances,laws, vehicle, behavior]). In a well-formed rule, if the predicate symbol of the conclusion is a member of a node TN (e.g., `evaluation`), the premises can only use predicate symbols from a subnode of TN (e.g., `places,circumstances,laws, vehicle, behavior`) or TN itself. So, for instance, it is not well-formed to conclude from the assurance contract of a vehicle's owner to the evaluation of the owner's parking behavior.

The topology graph, where the nodes represent sets of predicate symbols which can be premises of the supernodes, gives an overview of possible rules in an application domain. Figure 2 shows the topology graph for the traffic law domain.

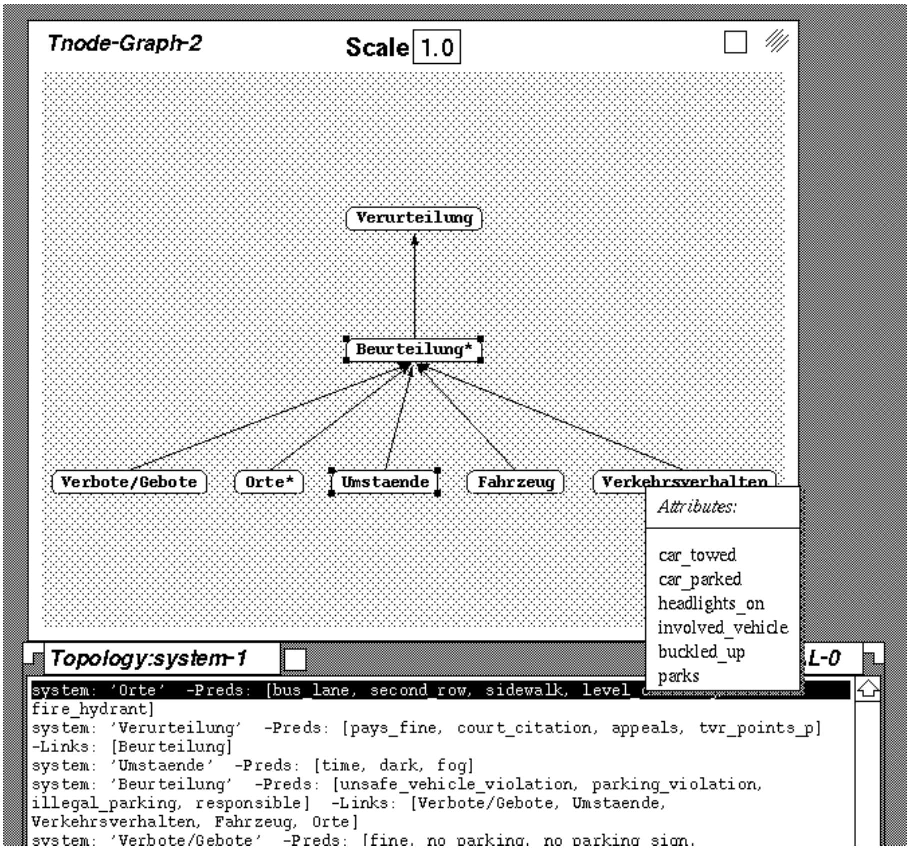

Figure 2: Topology graph

The topology graph can be viewed as a generalization of determinations (Davies, Russell 87). There, it is stated that a rule is sensible which relates some particular predicates. The topology generalizes this to sets of predicates: rules are sensible which use predicates of the same topology node TN or a predicate in TN for the conclusion, and predicates of subnodes of TN for the premises.

3.1.5 Rule Models

A rule model is a rule in which predicate variables are used instead of actual predicates of an application domain. A predicate variable can be instantiated by a predicate symbol of the same arity. There is a substitution Σ for predicate variables. Let RS be a rule model, then RSΣ is a (partially) instantiated one. If all predicate variables are substituted by predicate symbols, the rule RSΣ is predicate ground. Hence, a fully instantiated rule model is a rule. The rule model `R1(X,Y)& R2(Z,Y) --> Q(X,Z)` can be instantiated

Σ: `{R1/owner, R2/involved, Q/responsible }`

thus becoming our example rule. Rule models are ordered with respect to their generality such that the generality of fully instantiated rule models is given by theta-subsumption (Plotkin 70). RS1 is more general than RS2 iff for all Σ there exists a substitution of terms σ such that $RS2\Sigma \subseteq RS1\Sigma\sigma$. The above rule model is, for instance, more general than the rule model `R1(X,Y)&R2(Z,Y)&R3(X,Y) --> Q(X,Z)`, because every fully instantiated rule of the first one is a subset of the second one. [3] Σ is not allowed to replace different predicate variables by the same predicate symbol. Rule models are labeled by generated names such as, e.g. r1 or l2. The generality structure is presented as a graph, where the labels of rule models are the nodes and the generality relations are the links (see figure 3).

Rule models can also be partially instantiated. All possible instantiations of all rule models together form the *hypothesis space* for rule learning in MOBAL. The hypothesis space is structured by the generality structure of rule models. This is used by RDT (see below) in order to prune branches of rule models where no instantiation can lead to an accepted rule.

The following rule models can be used to model neighborhood relations. The domain predicate symbol `conn` states a neighborhood relation, the predicate variables `p0, p1, p2, p3, q` can be instantiated to characterize the related objects. The most general rule model is the statement of a two-place predicate which always is true. The next general rule model has in addition a one-place premise.

r0(q) : $\rightarrow$ q(x,y).

r1(p0,q) : p0(y) $\rightarrow$ q(x,y).

r2(p1,p0,q) : p0(y) & p1(y) $\rightarrow$ q(x,y).

r3(q) : conn(y, n0) $\rightarrow$ q(x,y).

r4(p0,q) : p0(y) & conn(y, n0) $\rightarrow$ q(x,y).

r5(p1,p0,q) : p0(y) & p1(y) & conn(y, n0) $\rightarrow$q(x,y).

r6(q) : conn(y, n0) & conn(y, n1) $\rightarrow$ q(x,y).

r7(p0,q) : p0(y) & conn(y, n0) & conn(y, n1)$\rightarrow$ q(x,y).

r8(p1,p0,q) : p0(y) & p1(y) & conn(y, n0) & conn(y, n1) $\rightarrow$ q(x,y).

r9(p0,q) : conn(y, n0) & p0(n0)$\rightarrow$ q(x,y).

[3]The more general rule model must be instantiated to become a subset of the more special one. This is the underlying meaning of theta-subsumption: a more general rule must be instantiated to become a subset of a more special rule.

r10(p1,p0,q) : p0(y) & conn(y, n0) & p1(n0) → q(x,y).

r11(p2,p1,p0,q) : p0(y) & p1(y) & conn(y, n0) & p2(n0)→ q(x,y).

r12(p0,q) : conn(y, n0) & conn(y, n1) & p0(n0) → q(x,y).

r13(p1,p0,q) : p0(y) & conn(y, n0) & conn(y, n1) & p1(n0)→ q(x,y).

r14(p2,p1,p0,q) : p0(y) & p1(y) & conn(y, n0) & conn(y, n1) & p2(n0)→ q(x,y).

r15(p1,p0,q) : conn(y, n0) & conn(y, n1) & p0(n0) & p1(n1)→ q(x,y).

r16(p2,p1,p0,q) : p0(y) & conn(y, n0) & conn(y, n1) & p1(n0) & p2(n1)→ q(x,y).

r17(p3,p2,p1,p0,q) : p0(y) & p1(y) & conn(y, n0) & conn(y, n1) & p2(n0) & p3(n1)→ q(x,y).

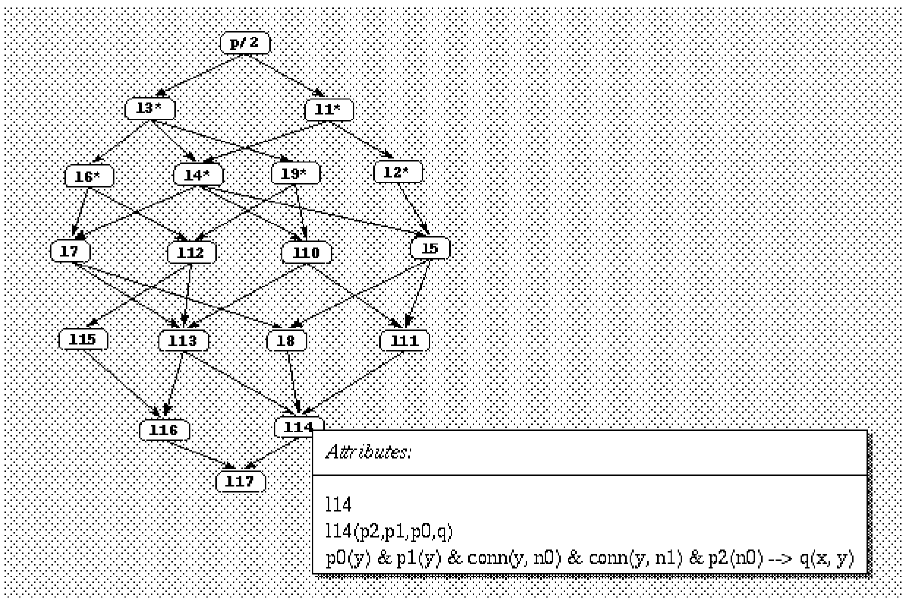

Figure 3: Generality structure of rule models

3.2 MOBAL's learning tools

The MOBAL system includes several learning tools:

- a rule discovery tool (RDT) which is a model-based, first-order logic learning algorithm inducing rules from facts
- a concept formation tool (CLT) which induces necessary and sufficient conditions for concepts from positive and negative examples
- a model acquisition tool (MAT) which abstracts rule models from rules
- a sort taxonomy tool (STT) which clusters constant terms occurring as arguments in facts

- a predicate structuring tool (PST) which abstracts rule sets to an overall structure of the knowledge base

To describe each of the learning tools in detail requires much more space than we have in this report. As we want to concentrate on the use of the tools - either by the user or by another tool - it is sufficient to describe them as black boxes and only indicate the principle of how they work.

3.2.1 RDT

The rule discovery tool RDT helps the user to find regularities in facts. The *task* is that of learning from observations or discovering regularities in order to predict new events.

Input: a set of facts, a set of rule models

Output: a set of rules which are most general inductive generalizations of the facts.

The necessary *input* to this model-based inductive algorithm are facts and rule models. It is not necessary that the facts are complete descriptions of examples. If rules are already learned or given by the user, they are taken into account by the algorithm. In particular, they are not re-discovered and they are not contradicted by a hypothesis for learning. Moreover, the inference engine performs forward inferences from (learned or given) rules, hence "saturating" the knowledge base for learning.

The learning *strategy* is top-down induction, i.e. the most general generalization is specialized until a rule is found which obeys a user-given acception criterion.

RDT can be *called* in different ways. It can be called with a time limit so that RDT learns within this CPU time limit and then stops. This allows to use RDT incrementally. The aim is that RDT can learn in the background during the modeling activity of the user. RDT then tries to learn about the predicate the user inputs as the predicate symbol of a fact. If the user wishes to focus on a particular predicate or a list of predicates, RDT looks for rules with these predicates in the conclusion. The set of predicates can be given by clicking on a node of the topology graph. RDT can also be called from CLT with a particular set of facts. The list of rule models is given by a parameter. The default is to use all rule models which are part of the domain model. The evaluation criteria for accepting a hypothesis can be set by the user. The basic building blocks for defining criteria are prepared.

A default setting is given. But, the user can define particular criteria and input them as parameter settings.

The *learning result* is a set of rules. The rules are not bound together in order to build sufficient and necessary conditions for concept membership nor is there an ordering of rules such as is in decision trees. Also, relations that hold between features are not distinguished from relations that hold between concepts or between features and concepts. The user may interpret the learned rules as characterizing concepts or as background knowledge. The learning result is used by the inference engine, hence, RDT performs *closed-loop learning*.

The basic idea behind this learning in predicate logic is to instantiate given rule models systematically and test the instantiations (i.e. rules) against ground facts. First, the most general rule model is instantiated. If an instance (i.e., a rule) is not accepted with respect to the acception criterion and still enough facts are available, the next special rule model is instantiated. This procedure is similar to Shapiro's refinement operator (Shapiro 81). But, whereas the refinement operator builds up a complete hypothesis space, the hypothesis space of RDT is restricted by the rule models. As the rule models regularly do not cover the forms of all possible Horn clauses they restrict the hypothesis space.

For a rule model each possible instantiation is tested. An instantiation is possible if the predicates which substitute predicate variables of the rule model have a compatible arity, sort restriction, and topology restriction. That is, the resulting rule hypothesis must be well-formed with respect to the sorts and the topology of predicates (see above). This restricts the hypothesis space further. For instance, the most general rule models can be: `P(X)-->Q(X), R(X,Y)-->Q(X,Y)` All 1-ary domain predicates with the same sort of an argument type which are in the same or linked topology node are tried as instances of the first rule model. If, for an instantiation Q/q and P/p there are many matching facts but not all of them justify the hypothesis, then the next special rule model is tried, e.g. `p(X)&R(X,Y)-->q(X)`. All compatible 2-ary predicates are tried as instantiations of R. Specializing hypotheses stops, if a rule already exists, becomes accepted, or if there are not enough facts that could match the (more special) hypothesis. As is easy seen, RDT is much quicker than e.g. FOIL(Quinlan 91) if the rule models are well-suited for the desired learning results. If a rule model is missing which would correspond to the desired result, RDT will not find the wanted rule. [4]

3.2.2 CLT

The concept learning tool CLT learns from positive and negative examples. The *task* is to define a concept on the basis of some concept instances

Input: a set of positive examples, a set of negative examples, a set of rule models

Output: a set of rules giving the sufficient and necessary conditions of a concept.

The new concept can serve as a feature for some other concepts. In other words: CLT can be used to construct new features. The *input* to CLT is a set of rule models, the name of the concept to be learned, and facts among which are those with the target concept name as predicate symbol. If this fact is positive, it contributes to a positive example. If this fact is negated, it contributes to a negative example. The concept can be a relational one, i.e. a two-place predicate can be defined by CLT. As is the case for RDT, also for CLT the user can input an acception criterion as a parameter of CLT. The list of rule models to be used by CLT needs not be identical with the list used by RDT.

CLT can be *called* by the user or by the knowledge revision module KRT [5]

The *learning result* is a set of rules which represent the sufficient and necessary conditions for concept membership. The sufficient conditions are rules with the concept in the conclusion. The necessary conditions are rules with the concept as a premise.

CLT uses the RDT algorithm. It is the focused use of RDT with the additional requirement of finding necessary conditions for the concept. [6]

[4] For a detailed description of RDT see (Kietz, Wrobel 91).

[5] Only the learning tools are described in this paper. The knowledge revision is a tool which handles contradictions, selects a rule to be deleted, or to be refined. The rule refinement is then performed either by the user or by the system. If a concept is missing which restricts the support set appropriately, KRT calls CLT to learn that concept.

[6] For a detailed description of CLT see (Wrobel 89).

3.2.3 MAT

The model acquisition tool MAT abstracts rule models from rules. The *task* is to generate rule models.

Input: a set of rules, a set of rule models

Output: new, non-redundant rule models.

As users prefer to input rules instead of rule models, the *input* to MAT are rules. The learning *strategy* is that of abstraction over rules. The rules are abstracted by turning predicate symbols from the application domain into predicate variables. It is checked, whether a new rule model corresponds to an already extisting one. If there are constant terms in the rule, these can be either turned into variables, too, or be introduced into the rule model. Rule models including a constant term as argument of a predicate may be of good use if the desired learning result is to clarify all properties and relations concerning a particular object or attribute value. The *result* is a rule model which is not redundant to any given one. [7]

3.2.4 STT

The sort taxonomy tool STT organizes the objects (constant terms) of an application domain into sorts and classes of sorts. The *task* is to structure the constant terms or objects of a domain. In other words, the task is to learn types for a typed logic. If the fact base of MOBAL changes and the non- incremental mode has been selected, then the user can call the update of the sort taxonomy.

Input: a set of facts

Output: a lattice of classes of sorts.

The *input* to the algorithm is a set of facts. The *output* of it is a lattice of classes of sorts. STT can be used either incrementally or as a single-step learner. The lattice gives an overview of the actual state of the fact base. It is used by the user for inspection and by the system to check the sort compatibility of new facts and rules (rule hypotheses). [8]

The learning *strategy* is that of bottom-up induction, where the learned classes are described by their extensions. The basic idea of the algorithm is

[7] For a detailed description of a previous version of MAT see (Thieme,89).

[8] For a more detailed description see (Kietz 88).

to produce sets of constant terms on the basis of their occurance at particular argument places of predicates. These sets are inspected with respect to subset relations, identity, or intersections. The sets are sort extensions. Equivalence classes are built for these sorts. The classes are organized in a lattice based on their subset relations or intersections. The most time consuming part of the algorithm is the calculation of intersections. The user can select whether intersections are to be built, or not. The algorithm is efficient, because it corresponds to learning in propositional logic.

3.2.5 PST

The predicate structuring tool PST organizes the predicate symbols of an application domain into linked sets of predicate symbols. The *task* is to structure the predicates of a domain.

Input: a set of rules

Output: an acyclic directed graph.

The *output* of it is an acyclic directed graph, the topology. The topology graph gives an overview of the rule base. It is used by the user for inspection and by the system to check the topology compatibility of rule hypotheses.

The learning *strategy* is that of abstraction over rule sets. The basic idea of the algorithm is to create a rule graph and then perform abstraction on it. A rule graph is a graph where the predicates of rule conclusions are in one node and the predicates of the premises are in its subnodes. As a predicate can only be in one node, the graphs for several rules can be combined easily to form the one rule graph for all rules of the rule base. This graph can be cyclic. It is transformed into an acyclic one by the first abstraction: for each cycle, a node is created with all the predicate symbols which occur in the cycle. The graph is further reduced by merging all nodes with the same successors or predecessors. In the *icterus* application the rule graph had 127 nodes for about 200 rules, the abstracted topology had only 50 nodes, thus giving a good overview of the rulebase. [9]

3.3 Cooperating learning tools

The brief description of MOBAL's learning tools already indicated their task and their use by user and system. The cooperation of the tools is

[9] For a detailed description of PST see (Klingspor 91)

- to use the results of another tool
- more particular, to call another tool
- to use the same knowledge as does another tool.

In MOBAL, RDT uses the results of MAT, STT and PST, if the user has not given the rule models, predicate declarations, or topology of predicates. In this way, the tools produce structures that on the one hand allow the user to inspect the evolving domain model. On the other hand, the tools produce prerequisites for another tool, namely RDT. Moreover, in doing so, the tools take the burden of structuring which otherwise would be on the back of the user. The results of STT and PST in particular illustrate the multi-functionality of represented knowledge. Figure 4 shows the interaction of the learning tools. The lines between tool names indicate the use of knowledge produced by a tool. The arrows denote a tool calling another one.

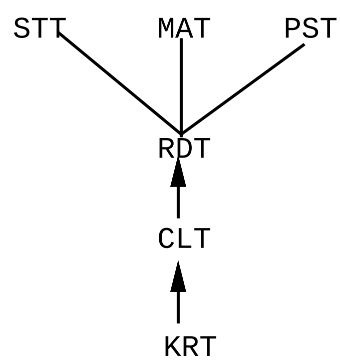

Figure 4: learning tools cooperating

CLT calls RDT with the name of the target concept and the list of rule models. CLT can be called by the user or by the knowledge revision. If a support set of a rule has too many exceptions (the criterion for "too many" given by the user or by default), CLT is called to define either a concept for the good rule applications or for the exceptions. The good and contradictive rule applications serve as positive and negative examples for CLT. In this way, the knowledge revision prepares the set of examples for CLT. The support sets are multi-functional in that they prohibit wrong inferences and can be used as examples for introducing new concepts.

The same knowledge, namely facts, is used by RDT, CLT and STT, where STT can use any set of facts and CLT needs positive and negative facts concerning a particular predicate, the target concept. Indirectly, via forward inferences (saturation), RDT, CLT, and STT use the rules also. In addition, RDT uses rules in order to not learn a known rule, again. If background knowledge were represented differently from the (learned) rules - as is the case in many learning systems - RDT could not use its learning results for further learning. MOBAL's uniform representation of background knowledge and learning results enables RDT to use new facts which were derived from learned rules as additional descriptions (examples). In particular, negated facts can be derived from learned rules with a negative conclusion and serve as counterexamples for further learning.

3.4 MOBAL cooperating with the user

The user is supported in modeling by several capabilities of MOBAL. As was pointed out above, we include inspection, testing (validation), and revisions in the modeling process. The tools of MOBAL serve the overall modeling activity of the user. In addition to the learning tools described above, there is an inference engine, a programmer's interface, a user interface, and a knowledge-revision tool. In the following, first the user interface is scetched. Then, it is shown how each knowledge item can be input by the user or inferred by a tool. Finally, the opprtunities of MOBAL for revising are indicated.

3.4.1 The interfaces

In general, all items are *input* using a edit-window named "scratchpad" where the user can edit the items before entering them into the system. A help window associated with the scratchpad shows the format of each item for input. If the data are available they can also be read into the system as text files in the scratchpad format. Using the programmer's interface which offers high-level system calls, MOBAL can be coupled with another system directly. Data can be exchanged between the systems using the commands of the programmer's interface.

All items can be displayed as texts in windows. The windows reflect changes of the knowledge base immediately. The content of a window can be *focused* so that only items containing a particular predicate symbol or constant term are displayed. Several windows can be opened for facts, rules, rule models, in parallel.

The windows are easily used for some *operations*. A double click with the mouse on a particular item pops up a menu where the user can select an operation (e.g.delete) or displaying the item *graphically*.

The user interface eases the inspection of the evolving domain model. But the overview, the consequences of changes, and the detection of contradictions are delivered by the tools and provide the real support for inspection.

3.4.2 Balanced adding of items

In this section, the balanced cooperation of system and user is described with respect to adding knowledge items. It is shown, that for each type of knowledge there exists a tool which creates items of this type and there is an interface which supports the user in adding items of this type.

The user may input *predicate declarations* with named *sorts*. This is sometimes useful, when it is easy to forget what argument type was supposed to occur where in a predicate. The predicate declaration then serves as a reminder of, e.g., where to put the person name in the predicate `owner`. If, however, the facts are already electronically available, the user needs not input predicate declarations. STT will do the job.

The user may input a *topology of predicates* in order to structure the domain model beforehand, e.g. with respect to steps of problem solving which uses the (learned) rules. [10] For instance, the leave nodes of the topology may consist of predicates which refer to the given data (observations) in an application. Intermediate nodes may refer to intermediate problem solving results. The root node may consist of predicates which refer to possible results of problem solving (possible solutions). In this way, the topology is a task structure for the performance element which uses the built-up knowledge base in an application. If, however, the user does not know the overall domain structure, PST can construct it on the basis of the rules.

The user may input *rules* and set the parameter such that MAT is called in order to obtain rule models from them. Or, the user may set the parameter to "direct rule input" so that MAT is not called for an inputed rule. The user may also input some rule models and call RDT for discovering rules. Thus, here again, there is a flexible work share of system and user.

[10] Learning serves the acquisition of a rule base for a particular application where the rules are put to use!

The user must input some *facts*. Facts are necessary for learning, inferring, and building the sort taxonomy. But, also facts can be added by the system's inferences. By selecting an inference depth for forward and for backward inferences (parameter of the inference engine), the user can force the inference engine to derive as many facts as possible within the selected inference depth (inference path length).

Hence, for each knowledge item there is a system tool adding it to the knowledge base, and there is the option that the user enters it. Balanced modeling is the flexible use of the tools for supporting the user to add items or to have the system adding items to the knowledge base.

3.4.3 Revisions

Revisions of all knowledge items are supported by MOBAL and the consequences are immediately propagated. If a rule or fact is deleted, all its consequences are deleted, too. Consequences are the facts derived from this rule or fact. Also updating the sort taxonomy and the abstracted topology reflects the change. It is not (yet) maintained, however, that a particular rule was learned because of facts that were deleted afterwards. This requires more book-keeping and would slow down the inference engine.

The interface allows to react to the displayed knowledge base. If, for instance, the user detects a (derived) fact which he wants to reject, he can either delete it in the fact window. Or, better, the user inputs this fact with the explicit negation. In this case, the negated fact serves as a constraint and influences learning. No rule covering the rejected fact can be learned any more. The knowledge revision detects contradictions of facts and displays graphically the inference pathes leading to the contradiction. The user or the system may perform the blame assignment and repair the rule base. Also, the explicit representation of exceptions in support sets and the call of CLT to form a new concept if too many exceptions of a rule have occurred helps to refine the domain model. In this way, MOBAL integrates inspecting, inputing, and revising a domain model.

4 Conclusion

There are some typical ways of using MOBAL. The extremes are to begin with facts and rule models and have the system learning rules, the sort taxonomy, and afterwards calling the topology tool. This is the "automatic mode". The other extreme is to begin with some known rules, declare the predicates, build up the topology, then input some facts and call the learning tools RDT or CLT. This is the "manual mode" where STT and PST are called for inspection purposes, the revision options of inference engine and knowledge revision are frequently used. Usually, modeling is performed using the system manually and automatically. The applications of MOBAL are

- traffic law domain - a self-made knowledge base with a rich structure and not so many facts; the knowledge base evolved in thee automatic mode.
- icterus - facts and rules were provided by Dr. Mueller-Wickop and the knowledge base was built up in the manual mode, using the tools for inspection, only.
- maldecensus testis - data were provided by the Foundation of Research and Technology, Hellas (FORTH); the data do not reflect the diagnosis model which was manually input by us in collaboration with a medical expert (Prof. Charisis).
- SPEED - knowledge about the supervision of security policy in distributed systems was provided by Alcatel-Alsthom Recherche, Marcoussis (AAR); the domain offers a rich structure where CLT successfully invented a new concept for rule refinement.

Except for the traffic law domain and the icterus domain, only preliminary studies have been performed. The collaboration with FORTH and AAR has just begun. However, some lessons have already been learned. Becoming acquainted with a system as complex as MOBAL takes some time. Setting the evaluation criteria, for instance, seems to be a skill which requires some experience with MOBAL. If users are familiar with attribute-value learning systems such as ID3, for instance, they tend to not input relations and not use all the options which MOBAL offers. In this case, the users have already done beforehand, what could have been learned using MOBAL. More naive users (with respect to computers) easier exploit the opportunities of MOBAL. The main advantage of MOBAL was the ease of inputing background knowledge or learning parts of the background knowledge. Users also employed the inspection and revision abilities of MOBAL. Moreover, MOBAL offers all advantages of a first order logic learning tool as opposed to a propositional logic one.

As a conclusion, MOBAL indeed accepts new items and integrates them into the knowledge base, supports the user in inspecting the knowledge base, detects contradictions, and refines the rules. All these tasks can be performed by the user or by a tool of the system. The users choose when to let the system do a task and when doing the task themselves. In both cases, the same knowledge representation and operations are applied. Therefore, MOBAL is a balanced cooperative system.

Acknowledgements

Work reported in this paper has partially been conducted within the project MLT which is funded by the ESPRIT programme of the European Community under P2154.
The MOBAL system is developed at the German National Research Center for Computer Science by (in alphabetic order) Joerg-Uwe Kietz, Volker Klingspor, Katharina Morik, Edgar Sommer, and Stefan Wrobel. It is a successor of the BLIP system which was developed at the Technical university Berlin. The author of this paper wishes to thank the colleagues from the Berlin as well as the colleagues from the Bonn days.

References

Davies, T.R. & Russell, S.J. (1987) A Logical Approach to Reasoning by Analogy. *Procs. of IJCAI-87*, Morgan Kaufmann.

Emde,W., Habel, C.& Rollinger, C.-R., (1983) The Discovery of the Equator or Concept- driven Learning. *Procs. of IJCAI-83*, Morgan Kaufmann.

Kietz, J.-U.& Morik, K. (1991) Constructive Induction: Learning Concepts for Learning. *Arbeitspapiere der GMD*, No.543

Kietz, J.-U.& Wrobel, S.(1991) Controlling the Complexity of Learning through Syntactic and Task-oriented Models. *Procs. of Int. Workshop Inductive Logic Programming*

Kietz, J.-U. (1988) Incremental and Reversible Acquisition of Taxonomies. Linster, M., Boose, J.& Gaines, B.(eds), *Procs. of EKAW-88*, GMD-Studien 143.

Klingspor, V. (1991) MOBAL's Predicate Structuring Tool. *Deliverable 4.3.2/G* of the MLT project, MLT-Report, No. GMD/P2154/22/1.

Kodratoff, Y.& Tecuci, G.(1989) The Central Role of Explanations in DISCIPLE. in K. Morik (Ed): *Knowledge Representation and Organization in Machine Learning*, New York: Springer, 1989.

Michalski, R.S. (1991) Inferential Learning Theory as a Basis for Multistrategy Task-Adaptive Learning. In R.S.Michalski & G.Tecuci (Eds): *First International Workshop on Multistrategy Learning*, West Virginia.

Morik, K. (1989) Sloppy Modeling. In K.Morik (Ed): *Knowledge Representation and Organization in Machine Learning*, New York: Springer, 1989.

Morik, K.& Kietz, J.-U. (1989) A Bootstrapping Approach to Conceptual Clustering. in A. Serge (Ed):*Procs. of 6th IWML*, San Mateo: Morgan Kaufmann.

Morik, K. (1991) Underlying Assumptions of Knowledge Acquisition and Machine Learning. *Knowledge Acquisition Journal, 3*, 137-156.

Morik, K., Causse, K.& Boswell, R. (1991) A Common Knowledge Representation Integrating Learning Tools. In R.S.Michalski & G.Tecuci (Eds): *First International Workshop on Multistrategy Learning*, West Virginia.

Quinlan, R. (1983) Learning Efficient Classification Procedures and their Application to Chess End Games. In R.S. Michalski, J.G.Carbonell & T. Mitchell *Machine Learning - An Artificial Intelligence Approach, Vol.I*, Palo Alto, CA:Tioga.

Quinlan, R. (1990) Learning Logical Definitions from Relations. *Machine Learning Journal,3*, 239-266, 1990.

Shapiro, E.Y. (1981) Inductive Inference from Facts. *Yale Research Report*, No. 192, Yale University.

Thieme, S. (1989) The Acquisition of Model Knowledge for a Model-driven Machine Learning Approach. In K. Morik (Ed.) *Knowledge Representation and Organization in Machine Learning*, New York: Springer, 1989.

Wrobel, S. (1987)Higher-order Concepts in a Tractable Knowledge Representation. In K. Morik(Ed) *Procs. German Workshop on AI*, Berlin,Heidelberg: Springer.

Wrobel, S. (1989) Demand-Driven Concept Formation. In K.Morik (Ed) *Knowledge Representation and Organization in Machine Learning*, New York: Springer.