Published in the *Proceedings of the KEML'97 Workshop on Knowledge Engineering Methods and Languages*

Noos: an integrated framework for problem solving and learning [*]

Josep Lluís Arcos Enric Plaza
IIIA, Artificial Intelligence Research Institute
CSIC, Spanish Council for Scientific Research
Campus UAB, 08193 Bellaterra, Catalonia, Spain.
{arcos,enric}@iiia.csic.es
http://www.iiia.csic.es

Abstract

One of the key issues in the current development of complex KBS is the necessity of incorporating learning capabilities to KBS. Specifically, the integration of learning components is considered as an essential topic for future KBS building and design. In this paper we present Noos as an integrated framework that supports problem solving methods and learning methods. A formalization of Noos using feature terms is presented. We explain the notion of episodic memory and its role in integrating learning and knowledge modelling. Finally, the integration of several eager and lazy learning methods is shown.

1 Introduction

Knowledge-level analysis of expert systems and the knowledge modelling frameworks developed for the design and construction of KBS are techniques for describing and reusing KBS components. These knowledge modelling frameworks, like KADS [2], CommonKADS [27] or Components of Expertise [21], are based on the task/method decomposition principle and the analysis of knowledge requirements for methods.

Machine learning techniques usually play an important role as knowledge acquisition tools in the process of knowledge modelling. Our approach is that certain knowledge acquisition tasks can be delayed and performed when the KBS is solving problems in the task environment. Following this approach, we have developed Noos as an integrated framework that supports problem solving methods (PSMs) and learning methods.

The delay of knowledge acquisition tasks implies that learning requirements have to be supported also by the KM framework. The requirements for incorporating learning in a KM framework are the *memory storage* ability, the *introspection* ability, and the *self-modification* ability. Concretely, a system that has to learn

[*]The research reported on this paper has been supported by the Spanish projects *ANALOG* CICYT-122/93 and *SMASH* TIC96-1038-C04-01, and by a CSIC fellowship for the first author. More information is available online at `http://www.iiia.csic.es/Projects/Noos.html`.

from its own experience has to be able to 1) inspect its own behavior (that has to be represented and stored in its memory), 2) analyze it and discover what aspects are responsible for a failure (or a success, or a delay, etc), and 3) decide how to transform itself (its knowledge, procedures for decision, etc) so that its future behavior is to be considered as improved.

Learning systems with memory storage and retrieval abilities need to represent part of its own problem solving behavior in the language of the system itself. We will call this kind of memory the *episodic memory* of the system. The form of this episodic memory is intimately bound to the scheme of representation used for inference. Noos is a representational framework where reasoning is represented in terms of tasks, methods that may achieve them, the subtasks needed to realize methods, and the knowledge or models used by these methods. In this approach, the episodic memory will store the decisions taken during the inference: successful methods engaged to tasks, results obtained by achieved tasks, methods that have failed to achieve tasks, etc.

Another important issue in integrating learning methods in Noos is the notion of *impasse*. Whenever there is a lack of knowledge directly usable by a problem solving method, an impasse arises. For instance, a generate and test method requires some knowledge to generate plausible hypotheses from problem descriptions: if this knowledge is lacking in the needed form, an impasse arises. In fact, the Noos language then generates a metalevel task, the task of solving that impasse [16]. Learning methods are then integrated as methods to solve the tasks generated by impasses in order to obtain the missing knowledge. In this way, using the knowledge modelling framework, we can analyze the knowledge requirements of a problem solving method (PSM) and, if this knowledge is not directly available, include some learning methods that may derive the knowledge required by that PSM. Thus, the role of a learning method is to generate knowledge in a form directly usable by a PSM.

We can summarize the role of learning methods in our framework as follows: a learning method is like a problem solving method with introspective capabilities such that 1) examines selected parts of the episodic memory (these selected parts are then considered "examples" or "cases" for learning) and 2) construct some new piece of knowledge needed to solve new problems.

The next section introduces the Noos approach to knowledge modelling and learning. In Section 3 feature terms are described as the formal basis of the language. Section 4 presents the integration of learning methods in our language and discuss the integration of different machine learning techniques as case-based reasoning, induction, and analytical learning. Finally, Section 5 contains the conclusions of the paper.

2 The Noos Approach

Knowledge-based problem solving is characterized by the intensive use of highly domain specific elements of knowledge. The purpose of knowledge modelling approaches is to describe this knowledge and how it is being used in a particular problem in an implementation independent way. Different knowledge modelling approaches have proposed different categories of knowledge elements and different abstractions to describe them.

We propose a model based on three knowledge categories: *domain knowledge*,

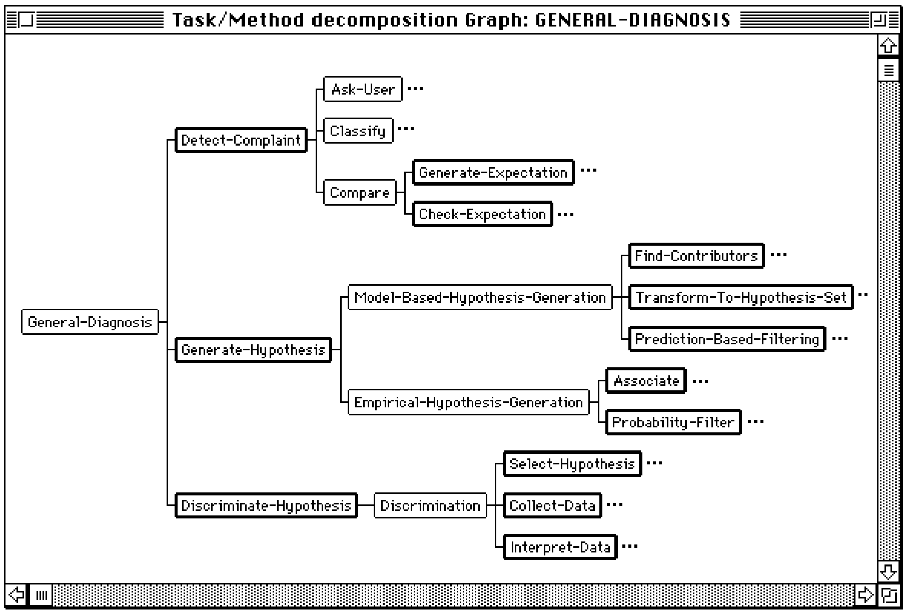

Figure 1: A browser of partial task/method decomposition for general diagnosis method. Tasks are drown with thin boxes and methods are drown with thick boxes.

problem solving knowledge, and *metalevel knowledge*. Moreover, we offer a mapping from this model to a representation language in order to provide a real computational framework to construct KBS.

2.1 The Noos Modelling Framework

The first knowledge category of the Noos framework is *domain knowledge*. The domain knowledge category specifies a set of *concepts* and a set of *relations* among them relevant for a given application. For instance, in the application of diagnosing car malfunctions, domain knowledge will be specified as a set of concepts capturing knowledge about cars or malfunctions. An example of a relation from cars to persons is the *"owner of a car"*.

Another category of the Noos framework is *problem solving knowledge*. Problems to be solved in a domain are modelled as *tasks*. For instance, following the previous example, the main task in the cars diagnosis domain is to establish car malfunctions. In our approach, *methods* model the ways to solve problems. Methods can be elementary or can be decomposed into subtasks. These new (sub)tasks can be achieved by corresponding methods in the same way. For a given task there may be multiple alternative methods (alternative ways to solve that task). This recursive decomposition of task into subtasks by means of a method is called the task/method decomposition. For instance, in Figure 1 the task/method decomposition of `general-diagnosis` method (following [6]) is shown. The `general-diagnosis` method is decomposed three subtasks, namely `detect-complaint`, `generate-hypothesis`, and `discriminate-hypothesis`. For each subtask one or several alternative methods are specified—e.g. subtask

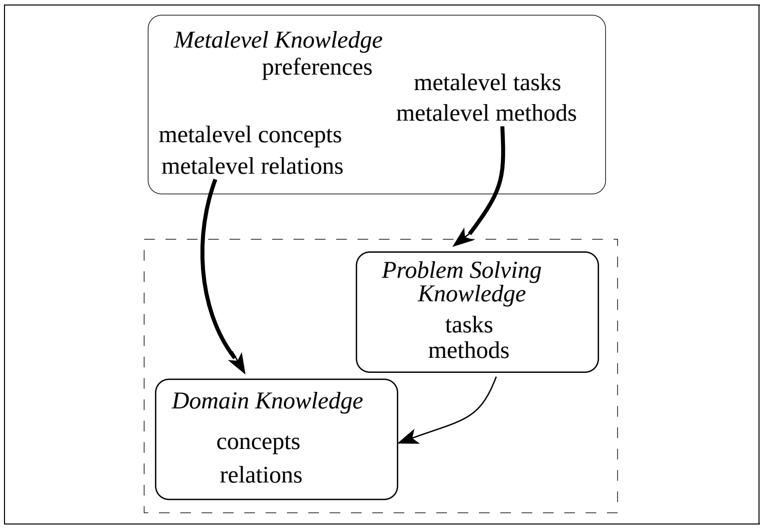

Figure 2: The Noos modelling framework.

`detect-complaint` has methods `ask-user`, `classify`, and `compare`. A relation can be described extensionally or intensionally. An intensional description of a relation can be modelled by means of *methods*. For instance, the age of a given person could be unknown but it is known that will be exactly the difference in years between the current date and the person's birthday.

The last category of the Noos framework is *metalevel knowledge*. Metalevel (or reflective) knowledge is knowledge *about* domain knowledge and problem solving knowledge. More specifically, metalevel knowledge can have models about concepts, relations, tasks, and methods. These models are formed by *metalevel concepts*, *metalevel relations*, *metalevel tasks*, and *metalevel methods* (see Fig. 2). Moreover, metalevel knowledge also includes *preferences* to model decision making about sets of alternatives present in domain knowledge and problem solving knowledge. For instance, metalevel knowledge can be used to model criteria for preferring some methods over other methods for a task in a specific situation. An example of metalevel task is one that chooses a method for a given task. An example of metalevel method is one that—for a specific situation—searches possible methods for a task, selects some methods as suitable alternatives, and finally ranks them using a set of preferences. This uniform representation of metalevel knowledge components by means of concepts, relations, methods, and tasks is the basis for the integration of learning.

Our purpose in the design of the knowledge categories of Noos was to use a set of knowledge categories close to the KADS [2, 27] and Components of Expertise [21] proposals. We are interested in proposing a compatible approach in order to take advantage of their work. Specifically, the research of Richard Benjamins on problem solving methods (PSM) for diagnosis [6] is able to help the design of problem solving methods in Noos. Nevertheless, there are some differences between Noos and the other proposals. The first difference is that since in our approach a PSM defines a way in which a task can be achieved, PSMs determine the subtask decomposition of tasks. In this sense we have join tasks and methods as elements of problem solving knowledge. Another difference is that task specification and method selection, as are understood in KADS, not exists as such, but may be modelled as metalevel knowledge in Noos.

Problem solving in Noos is considered as the construction of an *episodic model.* In this sense the Noos approach to problem solving is close to that of CommonKADS [27] and the TASK language [14]. The view of "problem solving as modelling" is that problem solving is the construction of an episodic model from problem data and problem solving knowledge. A clear and explicit separation between tasks, methods, and domain knowledge permits the dynamical link between a given problem, tasks, and methods as well as the dynamical choice of a suited method to achieve a task in a given resolution context : a 'task' applies a 'method' on a 'episode' (described using domain knowledge and problem data). Thus, a episodic model gathers the knowledge pieces used for solving a specific problem. Once a problem is solved Noos automatically memorizes (stores and indexes) the episodic model that has been built. Episodic memory (see § 2.3) is the (accessible and retrievable) collection of the episodic models of the problems that a system has solved. The memorization of episodic models is a basic building block for integrating learning into a KM framework.

2.2 The Noos Language

Noos is an object-centered representation language based on *feature terms. Feature terms* are record-like data structures embodying a collection of *features.* Intuitively, a feature term is a syntactic expression that denotes sets of elements in some appropriate domain of interpretation. In this way feature terms can be viewed also as partial descriptions. Numbers, strings and symbols are considered as predefined feature terms without features. Feature terms are described formally in the next section.

All the knowledge elements of the Noos model are represented into the language by means of feature terms. This means that with a small set of computational elements we capture all the elements of the knowledge model. Besides, this uniform representation of the knowledge benefits the introspective capabilities of Noos.

Concepts, as defined by the Noos model, are mapped to the Noos language as feature terms. Relations are mapped to features. Specifically, a feature term representing a concept c embodies the set of features related to this concept c.

The notion of *refinement* is introduced as the methodology to define feature terms in the Noos language. Refinement involves two distinct aspects: *code reuse* and *subtyping.* On the one hand, a new feature term is constructed reusing another existing feature term: a new entity description N defined as a refinement of another entity description E includes all the feature descriptions defined in E not redefined in N. For instance, the `car` feature term can be defined with the common knowledge about cars. Then, specific models of cars can be defined by refinements of `car`. Finally, specific cars can be described as refinements of models of cars.

Methods are also mapped to the language as feature terms. The subtasks of a method are mapped to the language as features. Thus, the set of features defined in a method description is interpreted as the subtask decomposition of that method. This subtask decomposition of methods allows to define (sub)methods for each subtask in a uniform way. The Noos language provides a set of built-in methods. New methods can be defined from other existing methods by refinement. New methods can also be constructed as combinations of existing methods.

A detailed description of the Noos language is offered in [4].

Inference in Noos is on demand. Thus, inference starts when a user asks to solve a specific task by means of a *query expression* that engages such a task. When a task is *engaged* its corresponding method is evaluated. A method is decomposed into subtasks; when it is evaluated its subtasks are consequently engaged. Thus, the inference in Noos can be viewed as a chaining process along the tree of task/method decompositions. This recursive chaining ends when a method directly uses factual knowledge. A task is achieved when its corresponding method is successful, and a method is successful when all its required subtasks are achieved.

When no method is specified for a given task, an *impasse* occurs and the control of the inference is passed to the corresponding metalevel task. The metalevel task has to infer a partially ordered set of alternative methods for the originating task. This set of alternative methods may be given or may be inferred by means of a metalevel method. The partial order is interpreted as a preference order in the selection of a method for the task that originated the impasse. At the end of the inference of a task engaged by a query expression, the *combination* of the successful methods in the tree of task/method decomposition will be maximal with respect to the preference orders inferred by the metalevels tasks involved. The interpreter of Noos uses backtracking to search for a method combination maximal with respect to the preferences involved.

2.3 Episodic Memory

As we have seen, the episodic memory requires Noos to represent part of its problem solving behavior in the Noos language itself. The decisions taken during inference for a task are automatically stored in Noos. The set of such stored decisions constitute the episodic memory of a system. The storage of the task/method decomposition instantiated in the solution of a given problem will provide the case-specific model that may be used later in learning methods: successful methods engaged to tasks, results obtained by achieved tasks, and methods that have failed to achieve tasks. Moreover, using introspective methods new knowledge can be inferred: knowledge about achieved tasks (tasks with a successful method engaged to the task), failed tasks (tasks where all methods have failed to achieve the task), untried options (methods not tried while achieving a task).

Another required ability for incorporating learning mechanisms in Noos is the introspection capability. Noos has to provide a way for recalling and reusing past experience for solving new problems. That is, a way to examine the contents of episodic memory. Noos offers two ways to perform introspection: using metalevel methods and using a set of retrieval methods provided by the language. Metalevel methods provide a way to inspect specific portions of the episodic memory. This first kind of introspection assumes some knowledge about the set of useful episodes. On the other side, retrieval methods offer introspection capabilities when not using a set of fixed episodes. For instance, we can be interested in episodes where a specific task has been solved using facts and characteristics similar to the current problem. Since these requirements and the set of precedents only can be determined dynamically, retrieval methods are necessary for these purposes.

Noos provides a set of basic retrieval methods. Since retrieval methods are methods like any other built-in methods provided in Noos, new retrieval methods can be designed refining and combining the existing ones. Retrieval methods allow to inspect and analyse previous similar episodes. The similarity criteria are determined by specific domain knowledge about the relevance different features or requirements of problem solving methods.

3 Feature Terms

Our approach to formalize Noos is related to the research based on *ψ-terms* [1, 9], and extensible records [8, 10] that propose formalisms to model object-oriented programming constructs. As we have presented, Noos is an object-centered representation language based on *feature terms*. *Feature terms* are record-like data structures embodying a collection of *features*.

The difference between feature terms and first order terms is the following: a first order term, e. g. $f(x, g(x, y), z)$, can be formally described as a tree and a fixed tree traversal order—in other words, variables are identified by position. The intuition behind a feature term is that it can be described as a labeled graph—in other words, variables are identified by name (regardless of order or position). This difference allows to represent partial knowledge.

We describe the Noos signature Σ as the tuple $\langle \mathcal{S}, \mathcal{M}, \mathcal{F}, \leq \rangle$ such that:

- $\mathcal{S}$ is a set of *sort symbols* including $\perp, \top$;
- $\mathcal{M}$ is a set of *method symbols*;
- $\mathcal{F}$ is a set of *feature symbols*;
- $\leq$ is a decidable partial order on $\mathcal{S}$ such that $\perp$ is the least element and $\top$ is the greatest element.

We define an interpretation $\mathcal{I}$ over the signature $\langle \mathcal{S}, \mathcal{F}, \leq \rangle$ as the structure

$$\mathcal{I} = \langle \mathcal{D}^{\mathcal{I}}, (s^{\mathcal{I}})_{s \in \mathcal{S}}, (\ell^{\mathcal{I}})_{\ell \in \mathcal{F}} \rangle$$

such that:

- $\mathcal{D}^{\mathcal{I}}$ is a non-empty set, called *domain* of $\mathcal{I}$ (or, universe);
- for each symbol s in $\mathcal{S}$, $s^{\mathcal{I}}$ is a subset of the domain; in particular, $\top^{\mathcal{I}} = \mathcal{D}^{\mathcal{I}}$ and $\perp^{\mathcal{I}} = \emptyset$;
- for each feature ℓ in $\mathcal{F}$, $\ell^{\mathcal{I}}$ is a total unary function $\ell^{\mathcal{I}} : \mathcal{D}^{\mathcal{I}} \mapsto \mathcal{P}(\mathcal{D}^{\mathcal{I}})$. When the mapping is not defined it is assumed to have value $\top$.

Methods are interpreted as functions (see Section 3.3).

Given the signature Σ and a set $\mathcal{V}$ of variables, we define *feature terms* as follows:

Definition 1 *A feature term ψ is an expression of the form:*

$$\psi \quad ::= \quad X : s\, [f_1 \doteq \Psi_1 \cdots f_n \doteq \Psi_n]$$

where X is a variable in $\mathcal{V}$, s is a sort in $\mathcal{S}$, $f_1, \cdots, f_n$ are features in $\mathcal{F}$, $n \geq 0$, and each Ψ_i is either a feature term or a set of feature terms.

Note that when $n = 0$ we are defining only a sorted variable $(X : s)$. We call the variable X in the above feature term the *root* of ψ, and say that X is *sorted* by the sort s (noted $Sort(\psi)$) and has features $f_1, \cdots, f_n$.

Using this syntax for feature terms, the following expression

$$X : Person \left[\begin{array}{ll} lastname & \doteq \text{Smith} \\ drives & \doteq Y : Car \left[\begin{array}{ll} owner & \doteq X \\ model & \doteq Z : Ibiza \end{array} \right] \end{array} \right]$$

is an example of a feature term denoting persons with a lastname Smith, who drive a car. Moreover, these persons are the owner of the car that drive and the model of the car is Ibiza.

A feature term is a syntactic expression that denotes sets of elements in some appropriate domain of interpretation ($[\![\psi]\!]^{\mathcal{I}} \subset \mathcal{D}^{\mathcal{I}}$). Thus, given the previously defined interpretation $\mathcal{I}$, the denotation $[\![\psi]\!]^{\mathcal{I}}$ of a feature term ψ, under a valuation $\alpha : \mathcal{V} \mapsto \mathcal{D}^{\mathcal{I}}$ is given inductively by:

$$[\![\psi]\!]^{\mathcal{I}} = [\![X : s[f_1 \doteq \psi_1 \cdots f_n \doteq \psi_n]]\!]^{\mathcal{I}} = \{\alpha(X)\} \cap s^{\mathcal{I}} \bigcap_{1 \leq i \leq n} (\ell_i^{\mathcal{I}})^{-1}([\![\psi_i]\!]^{\mathcal{I}})$$

where $f^{-1}(S)$, when f is a function and S is a set, stands for $\{x | \exists S' \supset S$ such that $f(x) = S'\}$; i.e., denotes the set of all elements whose images by f contains at least S.

Using this semantical interpretation of feature terms, it is legitimate to establish a relation order between terms. Given two terms ψ and ψ', we will be interested in determine when $[\![\psi]\!]^{\mathcal{I}} \subset [\![\psi']\!]^{\mathcal{I}}$.

3.1 Subsumption

The semantical interpretation of feature terms brings an ordering relation among feature descriptions. We call this ordering relation as *subsumption*. The intuitive meaning of subsumption is that of *informational ordering*. We say that a feature term ψ_1 subsumes another feature term ψ_2 ($\psi_1 \sqsubseteq \psi_2$) when all information in ψ_1 is also contained in ψ_2.

Definition 2 (Subsumption)
Given two feature terms ψ and ψ', ψ subsumes ψ', $\psi \sqsubseteq \psi'$, when :

1. *$Sort(\psi) \leq Sort(\psi')$, and*
2. *for every $f_i \doteq \Psi_i$ defined in ψ, $\forall \psi_i \in \Psi_i$ then f_i has to be defined in ψ' as $f_i \doteq \Psi'_i$ and $\exists \psi'_i \in \Psi'_i$ such that $\psi_i \sqsubseteq \psi'_i$.*

Notice that this definition of subsumption provides a concrete interpretation of the subsumption between two sets: given two subsets of terms u, v we say that $u \sqsubseteq v$ if the terms provided in u are extended and refined in v; formally,

Definition 3 *Given two subsets of terms u, v we say that $u \sqsubseteq v$ if for each $x \in u$, there is a $y \in v$ such that $x \sqsubseteq y$.*

This subsumption notion between sets corresponds to the *lower powerdomain* interpretation of sets [11].

For instance, the previous presented feature term is subsumed by the next feature term denoting persons driving a car with an owner and any car model:

$$X : Person \left[\; drives \quad \doteq Y : Car \left[\begin{array}{ll} owner & \doteq\; V : Person \\ model & \doteq\; Z : Car\_model \end{array} \right] \right]$$

Finally, we introduce the notion of *equivalence* among feature terms:

Definition 4 (Equivalence)

Given two feature terms ψ and ψ', we say that they are syntactic variants if and only if $\psi \sqsubseteq \psi'$ and $\psi' \sqsubseteq \psi$.

3.2 Understanding feature terms as clauses

Feature terms can be also understood as conjunctions of clauses. This clausal representation is useful and more usual for explaining learning methods. There are two kinds of atomic clauses: sort clauses $(X : s)$ and feature clauses $(f(X,Y))$. A given feature can be represented also as a conjunction of these two kind of atomic clauses.

Thus, we associate each feature term $\psi = X : s[f_1 \doteq \psi_1 \cdots f_n \doteq \psi_n]$ with a clause $\phi(\psi)$ as follows:

$$\phi(\psi) = X : s \wedge f_1(X, Y_1) \wedge \phi(\psi_1) \wedge \cdots \wedge f_n(X, Y_n) \wedge \phi(\psi_n)$$

where $Y_1, \cdots, Y_n$ are roots of $\psi_1, \cdots, \psi_n$ respectively.

For instance, the first feature term presented as example is represented as a clause in the following way:

$$\begin{array}{lllllll} X : Person & \wedge & lastname(X, \text{Smith}) & & & & \\ & \wedge & drives(X,Y) & \wedge & Y : Car & \wedge & owner(Y,X) \\ & & & & & \wedge & model(Y,Z) \;\wedge\; Z : Ibiza \end{array}$$

3.3 Representing methods as feature terms

Methods are represented by means of *evaluable* feature terms. The features of a given method indicates either a reference to some knowledge source required by the method, or a subtask the method requires to be accomplished. Each (sub)task has to be achieved by a corresponding (sub)method. In turn, (sub)methods are represented by means of new *evaluable* feature terms. The evaluation of a method performs a specific combination of the knowledge sources and the results of the subtasks returning a value as a result. The value returned can be a number, a string, a symbol, or any other feature term—including a method. Noos provides an initial set of built-in methods. Each built-in method has a set of built-in features. For instance, `Identity?` built-in method is a comparison method that expects the first element to compare as the feature value of its feature `item1` and the second element in `item2`. Examples of Noos built-in methods are arithmetic operations, set operations, logic operations, operations for comparing feature terms and other basic constructs such as conditional or sequencing. The semantics of conditional and sequencing is the usual in declarative programming languages and they embody the only control constructs in Noos. New methods are defined by refinement and combination of built-ins or other already defined methods.

Methods are conceived of as functions the parameters of which are passed by name (the required feature names) instead of position. An advantage of this representation is that evaluation is not strict, since parameters of a method can be passed in any order to the method. Formally, a method $m \in \mathcal{M}$ is interpreted as a function

$$m : (f_1, \Psi_1) \times \cdots \times (f_n, \Psi_n) \mapsto \Psi$$

where $f_1, \cdots, f_n$ are feature names, $\Psi_1, \cdots, \Psi_n$ are their corresponding actual values, and Ψ is the result of the method.

An example of a Noos built-in method is the `conditional` method that has three subtasks given by required features with name `condition`, `result`, and `otherwise`. Below the `conditional` method is shown (being a refinement of `method` indicates it is a built-in).

$$conditonal : method \begin{bmatrix} condition & \doteq boolean \\ result & \doteq \top \\ otherwise & \doteq \top \end{bmatrix}$$

The `conditional` method performs first the subtask `condition` and depending on its result being `true` or `false`[1] either the `result` subtask or `otherwise` subtask is performed and the value obtained is returned as the result.

In order to describe the corresponding method of a given subtask the syntax for feature terms is extended. We use the # token as a way to introduce methods engaged to subtasks: a given method m for a subtask with name f_i is described using the $f_i \doteq \#m$ syntax. Introducing the # token we can differentiate methods from references for subtasks. For instance, a generate and test method applied to a diagnosis task, being a refinement of `sequence` built-in method, is decomposed in two subtasks called `generate` and `test` and these two subtasks can be achieved respectively by specific generate and test methods:

$$G\&T : sequence \begin{bmatrix} generate & \doteq \#generate\_method\_4 \\ test & \doteq \#test\_method\_7 \end{bmatrix}$$

We say that a method is *closed* when all the needed features (references and subtasks) are specified. Only closed methods can be evaluated and return a value. Closed methods can be used as ways to infer feature values. Specifically, introducing a closed method we describe a feature value by means of a task instead of a fixed value. That is to say, using the syntax $f \doteq \#m$ in some feature term we are specifying that method m will be used to solve task f. Clearly, this syntax is the same as in subtask decomposition of methods. For instance, in the domain of electronic circuits an `adder` component can be represented as a feature term with two input ports (represented by the features `A_in` and `B_in`), one output port (represented by the `out` feature), and three wires . The value of the `voltage` feature of the output port can be expressed as the addition (`add`) of the two input ports as follows

[1]Both `true` and `false` are the boolean constants in Noos.

$$
\mathit{Adder} : \mathit{block} \left[\begin{array}{ll}
\mathit{A\_in} & \doteq X : \mathit{port} \left[\begin{array}{ll} \mathit{voltage} & \doteq X_1 : \mathit{number} \\ \mathit{wire} & \doteq X_2 \end{array} \right] \\
\mathit{B\_in} & \doteq Y : \mathit{port} \left[\begin{array}{ll} \mathit{voltage} & \doteq Y_1 : \mathit{number} \\ \mathit{wire} & \doteq Y_2 \end{array} \right] \\
\mathit{out} & \doteq Z : \mathit{port} \left[\begin{array}{ll} \mathit{voltage} & \doteq \#Z_1 : \mathit{add} \left[\begin{array}{ll} \mathit{item1} & \doteq X_1 \\ \mathit{item2} & \doteq Y_1 \end{array} \right] \\ \mathit{wire} & \doteq Z_2 \end{array} \right] \\
\mathit{A\_wire} & \doteq X_2 : \mathit{wire} \\
\mathit{B\_wire} & \doteq Y_2 : \mathit{wire} \\
\mathit{out\_wire} & \doteq Z_2 : \mathit{wire}
\end{array} \right]
$$

4 Learning and Knowledge Modelling

Machine Learning (ML) techniques have been used by Knowledge Modelling methodologies as a way to acquire certain models in the knowledge acquisition (KA) process conducive to building a KBS. Our interest is in developing KBS with *integrated* learning capabilities. This option means essentially that certain knowledge acquisition tasks are *delayed* from the KBS design and construction phase to the phase in which the KBS system is actually used in the task environment. Since KM methodologies views KA as a process that basically build models, our approach means that some models are not built[2] in the first phase, and their construction is delayed to the second phase where appropriate ML methods are appointed to generate those models.

This delay of KA tasks also implies the following:

- Knowledge Modelling of the KBS has to include modelling of KA goals
- ML techniques have to be modelled inside the framework, in our case ML techniques are modelled as methods
- knowledge requirements of ML methods have to be addressed; in our framework *episodic memory* is used to model the specific requirement of modelling the "examples" or "cases" used by ML techniques.

4.1 Integration of machine learning methods

Integrated learning is modelled as a process with three main subtasks: *Introspection*, *Construction*, *Revision.* This scheme allows us to model different ML methods and their integration into a general problem solving system by developing specific methods for the three main subtasks. Let us consider these tasks in turn:

Introspection This task is the process by which past experience (episodic memory of the system itself or provided by a teacher) is accessed, selected and retrieved for purposes of new problem solving. In simple situations this task may merely select a subset of examples provided. In complex situations the system may have to decide which (sub)parts of all the episodic memory qualify as "examples", i. e. are interesting to learn from.

[2] Or, in general, a preliminary model is build but needs to be improved.

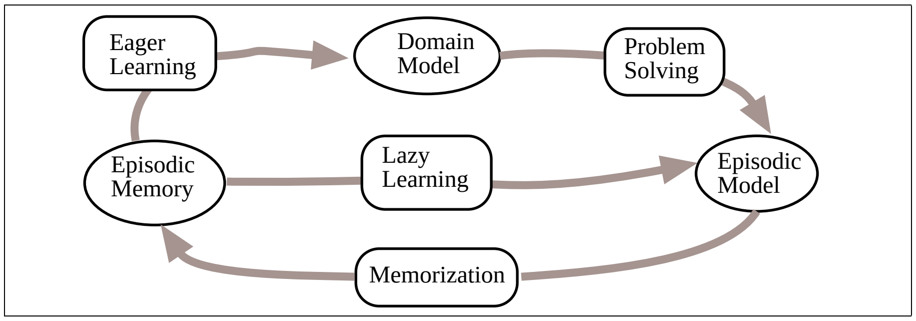

Figure 3: Lazy and eager learning and the construction of domain models and episodic models.

Construction This task uses the relevant examples (cases) to generate some new model or body of knowledge. Eager and lazy ML methods (see below) differ in the nature of what they construct.

Revision This task decides whether and how the system knowledge is modified by the newly constructed model. In simple situations the new model just satisfies a knowledge requirement or substitutes the old model. In more complex cases, the task has to estimate whether the new model does improve the overall performance of the system (e. g. preventing overfitting or "expensive" rules).

Construction task is what is usually called a *learning algorithm*. This is because in off-line learning the "introspection" task is simply done by a human engineering the system, while revision is present only in incremental algorithms—and in interactive systems decisions about revision are taken by the human engineering the system. These human-intensive processes are modelled in our framework as methods for the introspection and revision tasks that interact with the human expert/engineer.

Before proceeding to review how different ML methods are integrated into the Noos framework, it is useful to distinguish between learning methods of eager and lazy nature.

Eager learning Past experience is used *in toto* to provide a new model or body of knowledge to be used for a specific problem solving method that will be applied in all future problems (of a specific kind). The paradigmatic eager learning methods are inductive techniques that generate abstract knowledge from specific examples and teacher input. In non-incremental approaches, past experience can be disposed of when the new model has been generated.

Lazy learning Past experience is accessed, selected and used in a problem-centered approach. The paradigmatic example is CBR, where for each new problem the system filters out irrelevant past experiences, and focuses on the relevant part from which it extracts or generates new knowledge to the extend needed for solving that particular problem. We view lazy learning as constructing an *episodic model* for the current problem—instead of constructing a generic model[3].

[3]The generic domain model is only needed for being used in constructing episodic models while solving future problems.

Lazy learning algorithms differ from others in that they delay inference and that they are problem-centered. Thus, they generally have low computational costs during training and high costs during testing. Another difference is that eager ML methods try to optimize on the *average* outcome for the future (unseen) problems based on the assumption that the past (seen) solved problems are a representative sample of the problems appearing in the task environment. Lazy ML methods may in principle incur on higher runtime costs—that should be nonetheless practicable for the task environment—but can optimize performance on a problem by problem basis.

4.2 Case-based Reasoning

Case-based reasoning (CBR) forms a family of techniques and systems that integrate lazy learning with problem solving where domain-specific knowledge and methods are used. We model CBR in Noos as *case-based methods*. It is clear that case-based methods can be integrated in our framework because of the notion of memory: past problem solving episodes are stored in memory and can be recalled using the retrieval methods. These stored problem solving episodes constitute the set of precedents (also called cases) for case-based methods. Generally speaking, case-based methods are decomposed into three subtasks: **`retrieve`**, **`select`**, and **`reuse`**. Since there are several possible methods usable in each subtask, several CBR techniques can be integrated in this way.

The first **`retrieve`** subtask requires a method that recovers previous solved cases from the episodic memory using a relevance criterion. The goal of the `select` subtask is to rank the cases obtained in the retrieval subtask according to domain criteria. Since cases are represented as feature terms, introspection during **`retrieve`** subtask is expressed in Noos as operations over the memory of feature terms representing past cases. Several basic *retrieval methods* are provided by the language and they can be seen as queries to the episodic memory of a system. For instance, **`retrieve-by-pattern`** is a retrieval method based on the subsumption relation defined in Section 3.1. This retrieval method selects all feature terms in episodic memory that are subsumed by a feature term called *pattern*. If the pattern is the following,

$$X : Car \left[owner \quad \doteq Y : Person \left[\begin{array}{ll} Nationality & \doteq Z : Andorran \\ model & \doteq W : Ibiza \end{array} \right] \right]$$

then **`retrieve-by-pattern`** retrieves from episodic memory all Ibiza cars whose owner is a citizen of Andorra. More complex retrieval methods are described in [15, 17].

Finally, the method engaged to the **`reuse`** subtask will construct a solution for the new problem—an episodic model for that problem. Usually in CBR the episodic model is built either taking the solution given in the most relevant precedent or constructing a new solution by adapting the solution(s) of one or more precedents. A usual method for **`reuse`** subtask is what Carbonell called *derivational reply*. In the Noos language derivational reply is automatically supported and consists in the following:

1. Once a most relevant precedent case is chosen by the *select* subtask, the method used to solve the same task the system is now involved in is accessed.

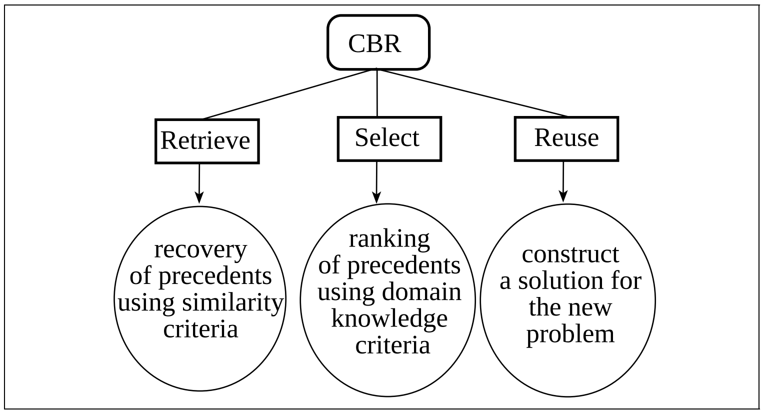
CBR
Retrieve
Select
Reuse
recovery of precedents using similarity criteria
ranking of precedents using domain knowledge criteria
construct a solution for the new problem

Figure 4: Task/method decomposition of Case-based reasoning methods.

2. The method is copied and re-instantiated to the current problem — i. e. method references are mapped from the past case to the current case and bound to the feature term of the current problem.

3. The new method is used to solve the current task.

The derivational reply method can be used not only in planning tasks, as proposed by Carbonell [26], it is a completely general method that can be used, in principle, in any component-based framework supporting component reuse.

4.3 Inductive learning

Induction also can be modelled in Noos by methods. The goal of an inductive method is to generate the knowledge needed by a problem solving method. Induction generalizes from a set of problem solving episodes. In general, the ML community defines induction as a process that constructs a general description that "generalizes" (in some sense that may vary[4]) the positive examples—and does not generalize the negative examples. In our framework feature terms offer a representation formalism that is a subset of first order logic where *subsumption* provides a well defined and natural way for defining generalization relationships:

$$\psi \text{ is more general than } \psi' \text{ iff } \psi \sqsubseteq \psi'$$

An example of the use of an inductive method is the generation of class description for a category or concept from a set of examples. The acquired knowledge will be used by an identification method deciding whether or not new examples pertain to a certain category. In general inductive methods can be characterized as search methods that follow certain *bias*: constrains upon the search space effectively searched and strategies for searching certain subspaces before others. These bias of ML methods are similar to assumptions for PSM, e.g. a ML inductive method can be exhaustive (or complete—if it assures it will find a generalization if it exists) or not exhaustive. However this comparison is left for future work.

[4]Specially in ILP (induction of logic programs) several different semantics have been proposed for the notions of generalization and subsumption.

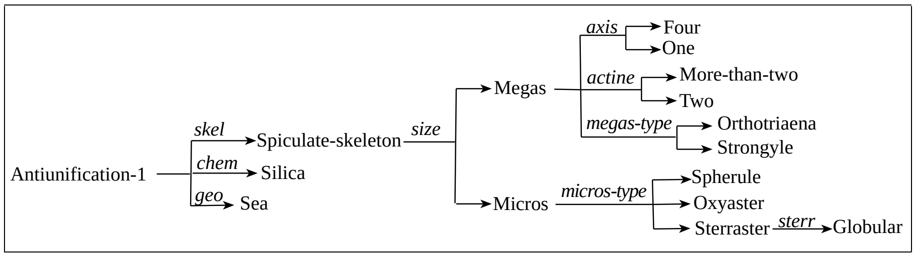

Figure 5: An example of induction by antiunification in the domain of marine sponges.

The induction methods currently developed in Noos are based on the *antiunification* and the *subsumption* operations of Noos[15]. The antiunification of a set of descriptions gives a new description that is a most specific *generalization* of them. Formally, the antiunification of a set of descriptions yields a greatest lower bound with respect to subsumption ordering. Inductive methods are designed in Noos using antiunification and specific biases. For instance, using relevance measures of attributes we can establish a bias for the generation of disjunctive descriptions of concepts. Figure 5 shows a generalization of two marine sponges using the antiunification method [17].

For the moment Noos inductive methods work only on descriptions and not on methods. In other words, Noos inductive methods can build domain models by generalization of episodic models. What is yet future work is learning of programs (methods) from examples (as in inductive logic programming), but analytical learning of methods has been integrated in Noos as shown in the next section.

4.4 Analytical learning

Analytical learning techniques also can be modelled in Noos by methods. The goal of analytical learning methods (or EBL-like methods) is to construct a new PSM for a given task from a training example solved by a PSM. The new PSM has to obey some conditions of operationality determined by the domain model. The training example is a problem for which the PSM is able to find a solution. This learning approach needs to inspect the so called explanation (or trace) of the inference performed by the PSM for the training example. In Noos the task/method decomposition instantiated in the solution (the episodic model) of the training example constitute the explanation (proof) of the solution. Thus, analytical learning methods in Noos are metalevel methods that given an episodic model built by a PSM m (1) inspect the methods that succeeded in each subtask of the task/method decomposition tree of m, and (2) construct a new PSM for the task according to an operational criterion.

We have developed PLEC, an EBL-like learning method that constructs new operational problem solving methods according to operational criteria determined for each specific problem. For instance, the following is a definition of a PSM that determines whether or not an object (passed as a parameter in the `source` feature) is an example of a `cup`:

$$cup\_method : conjunction \begin{bmatrix} source & \doteq Y : Cup\_33 \\ item1 & \doteq Y.stable \\ item2 & \doteq Y.liftable \\ item3 & \doteq Y.open\_vessel \end{bmatrix}$$

where an expression like $Y.f$ is interpreted as a reference to the feature f in Y (`cup_33` in this example) following the common dot notation for field selection in records. This kind of references are out of the scope of the paper but interested reader can find in [4] a detailed description of references in Noos.

Training instance `cup_33` is defined with the following problem data:

$$Cup\_33 : cup \begin{bmatrix} owner & \doteq John \\ light & \doteq true \\ color & \doteq red \\ handle? & \doteq true \\ bottom & \doteq Y : bottom \begin{bmatrix} flat & \doteq true \end{bmatrix} \\ concavity & \doteq X : concavity \begin{bmatrix} upward\_pointing & \doteq true \end{bmatrix} \end{bmatrix}$$

The application of the `cup_method` engages in turn other PSMs for determining when a given object is `stable`, `liftable`, and `open_vessel`. Using the episodic model built after a specific problem is solved, PLEC constructs a new operational method for determining when an object is an example of a cup. The operational criterion of PLEC is that the constructed method refers only to features present in the problem (e.g. those present in `Cup_33` in above). Thus, the new method built by PLEC will be directly applicable to the problem and skip intermediate inferences. The result obtained by PLEC for the previous example is the following operational method for cup:

$$X : conjunction \begin{bmatrix} source & \doteq Y : \top \\ item1 & \doteq Y.bottom.flat \\ item2 & \doteq Y.handle? \\ item3 & \doteq Y.light \\ item4 & \doteq Y.concavity.upward\_pointing \end{bmatrix}$$

5 Conclusions and Related work

In this paper we have presented Noos as an integrated framework that supports problem solving methods and learning methods. The integration of learning in a KM framework implies that new requirements have to be supported in the framework. The first requirement is *episodic memory*: the necessity to store the episodic models of the problems that the system solves in order to learn from its own experience. These episodic models have to be amenable to be *inspected* and analised by learning methods. Finally, the new knowledge generated by learning methods has to be *incorporated* in the framework modifying the future behavior of the system.

Our work is related to cognitive architectures that integrate learning with problem solving like SOAR [13], THEO [12], and PRODIGY [7]. SOAR learning is based on a single method called *chunking* while our purpose in Noos has been to integrate multiple learning methods within a problem solving framework. THEO integrates multiple learning methods but provides more restricted metalevel capabilities than

Noos—the metalevel methods allowed are predefined and ranked by a total order. Regarding PRODIGY, problem solving is based on a state-space search planning engine. Each learning method is an external module that learns from the planning engine performance, and that modifies this engine according to the learning results. These learning modules are external software packages that communicate to and from the planning engine using a uniform read/write protocol to the Prodigy database where all types of rules—inference rules, operators, and control rules—are represented uniformly.

Related work on knowledge modelling frameworks includes the ComMet framework [21], the KADS and CommonKADS methodologies [2, 27], and the Protégé-II system [18]. Our purpose in the design of the knowledge components of Noos was to use a set of knowledge components close to the KADS and ComMet proposals. KADS reflective framework, called "knowledge-level reflection" [25], specifies the system self-model of structure and process, very much like our metalevel model of domain knowledge, tasks, and methods allows Noos to have a self-model.

In the Protégé-II system there is a difference in implementation of the support system: mechanisms, the basic building blocks in Protégé-II, are implemented in Lisp and thus new mechanisms require new programs in Lisp. The philosophy of Noos is different: methods can be decomposed in a finer grain into elementary subtasks that use a set of elementary methods (e.g. conditionals, set intersection, etc.) provided by Noos.

The research on problem solving methods (PSM) like [6, 23] is focussed on acquiring a wide library of components and the reusability of them in several applications. This work is very related to Noos methods. The construction of libraries is a complementary work not tackled in Noos. We plan to explore the incorporation of reasoning about method applicability and reusability as a metalevel knowledge in Noos.

Related work on the use of knowledge level models to describe learning methods are [24], describing EBL methods, and [20], describing decision tree induction methods and implementing them in KresT (the workbench of the ComMet framework). Autognostic and Meta-Aqua are other related systems that use learning by reflecting on problem solving. Autognostic constructs structure-behaviour-function (SBF) methods for monitoring the problem solving in order to improve its efficiency and produce better solutions [22]. Unlike Noos, the Meta-Aqua approach is to use introspection for reasoning about failure [19].

We have used Noos to implement CHROMA [5], a system for recommending a plan for the purification of proteins from tissues and cultures. CHROMA learns from experience using two learning methods: CBR learning and induction. The reflective capabilities of Noos allow CHROMA to analyze and decompose problem solving and learning methods in a uniform way, and also to combine them in a simple and efficient way. SPIN is another system being developed using Noos at our Institute. SPIN is a sponge identification system for a marine sponges of the *Geodiidae* family. SPIN currently integrates a bottom-up induction method, a top-down induction method, and an CBR method based on an entropy measure [17].

A main concern of Noos as a representation language, has been to reconcile the notions of knowledge modelling (KM) developed in the recent years, with the principles of declarative programming languages. The result is that Noos offer a uniform representation for KM notions, inference (computation), and control.

Let us review alternative approaches to this issue. For instance, approaches to operationalize KADS have been proposed that augment the KM notions with some control constructs. These control constructs ultimately are the usual ones in programming languages: sequence, iteration, conditional, etc. However, this approach separates concerns of representation of expertise based on KM notions from "programming" constructs. Moreover, another necessity is defining the "atomic" inferences (or atomic computation elements) that are not part of the KM representation. For instance, the Protégé-II system implements atomic computation elements as chunks of Lisp code ("mechanisms"). New PSM may require implementing new "mechanisms". In summary, this "hybrid" approach use a programming language as an extension language in order to implement elementary inference or computation.

However, following the main trend in declarative programming languages, computation can be seen as inference. There is no need, in principle, for a three-level computational support to knowledge modelling: control superstructure, KM notions, and computational infrastructure. The Noos language is an example that this is possible. The reason why KM notions can be naturally embedded into Noos has two parts. First, feature terms offers a declarative blend of functional programming and object orientation. Object-orientation allows some KM notions to be represented in a natural way while functions also allows a natural way to deal with PSM. Thus, control constructs and atomic inferences are simply modelled as functions (Noos methods). Second, the reflective engine of Noos supports backtracking and a declarative form of control by means of preferences. Operationalizations of KM frameworks have mostly use rule-based implementations because the use of backtracking was important. Backtracking allows an under-specification of control, something not so easy on functional or object-oriented languages but that is supported by Noos. The original blend of functions and backtracking that Noos proposes is in the foundation of its capability to support KM in a way that is natural—once the user gets used to it.

References

[1] Hassan Aït-Kaci and Andreas Podelski. Towards a meaning of LIFE. *J. Logic Programming*, 16:195–234, 1993.

[2] H. Akkermans, F. van Harmelen, Guus Schreiber, and Bob Wielinga. A formalisation of knowledge-level model for knowledge acquisition. *Int Journal of Intelligent Systems*, 8:169–208, 1993.

[3] Josep Lluís Arcos and Enric Plaza. Integration of learning into a knowledge modelling framework. In Luc Steels, Guss Schreiber, and Walter Van de Velde, editors, *A Future for Knowledge Acquisition*, number 867 in Lecture Notes in Artificial Intelligence, pages 355–373. Springer-Verlag, 1994.

[4] Josep Lluís Arcos and Enric Plaza. Inference and reflection in the object-centered representation language Noos. *Journal of Future Generation Computer Systems*, 12:173–188, 1996.

[5] Eva Armengol and Enric Plaza. Integrating induction in a case-based reasoner. In J. P. Haton, M. Keane, and M. Manago, editors, *Advances in Case-Based Reasoning*, number 984 in Lecture Notes in Artificial Intelligence, pages 3–17. Springer-Verlag, 1994.

[6] Richard Benjamins. *Problem Solving Methods for Diagnosis.* PhD thesis, University of Amsterdam, 1993.

[7] Jaime Carbonell, C. A. Knoblock, and Steven Minton. Prodigy: An integrated architecture for planning and learning. In Kurt VanLehn, editor, *Architectures for Intelligence.* Lawrence Erlaum Associates, 1991.

[8] Luca Cardelli and John Mitchell. Operarions on records. In Carl A. Gunter and John C. Mitchell, editors, *Theoretical aspects of object-oriented programming: types, semantics and language design*, Foundations of computing series, pages 295–350. MIT Press, 1994.

[9] B. Carpenter. *The Logic of typed Feature Structures.* Tracts in theoretical Computer Science. Cambridge University Press, Cambridge, UK, 1992.

[10] Laurent Dami. *Software Composition: Towards an Integration of Functional and Object-Oriented Approaches.* PhD thesis, University of Geneva, 1994.

[11] C.A. Gunter and D.S. Scott. *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, chapter Semantic Domains. J. van Leeuwen, 1990.

[12] Tom Mitchell, J. Allen, P. Chalasani, J. Cheng, O. Etzioni, M. Ringuette, and J. Schlimmer. THEO: A framework for self-improving systems. In Kurt VanLehn, editor, *Architectures for Intelligence*, pages 323–356. Lawrence Erlaum Associates, 1991.

[13] Allen Newell. *Unified Theories of Cognition.* Cambridge MA: Harvard University Press, 1990.

[14] C. Pierret-Golbreich and E. Hugonnard. Organization and use of generic models based on the TASK language. In *EKAW'94*, 1994.

[15] Enric Plaza. Cases as terms: A feature term approach to the structured representation of cases. In Manuela Veloso and Agnar Aamodt, editors, *Case-Based Reasoning, ICCBR-95*, number 1010 in Lecture Notes in Artificial Intelligence, pages 265–276. Springer-Verlag, 1995.

[16] Enric Plaza and Josep Lluís Arcos. Reflection and analogy in memory-based learning. In *Multistrategy Learning Workshop MSL-93*, 1993.

[17] Enric Plaza, Ramon López de Mántaras, and Eva Armengol. On the importance of similitude: An entropy-based assessment. In *Third European Workshop on Case-Based Reasoning EWCBR-96*, Lecture Notes in Artificial Intelligence. Springer-Verlag, 1996.

[18] A. Puerta, J. Egar, S. Tu, and M. A. Musen. A multiple-method knowledge acquisition shell for the automatic generation of knowledge acquisition tools. In *the AAAI Knowledge Acquisition Workshop*, 1991.

[19] Ashwin Ram, Michael T. Cox, and S. Narayanan. An architecture for integrated introspective learning. In *ML'92 Workshop on Computational Architectures for Machine Learning and Knowledge Acquisition*, 1992.

[20] A. Slodzian. Configuring decision tree learning algorithms with krest. In *Knowledge level models of machine learning Workshop preprints*, 1994. ML-Net Familiarization workshops, Catania.

[21] Luc Steels. Components of expertise. *AI Magazine*, 11(2):28–49, 1990.

[22] Eleni Stroulia and Ashok K. Goel. Learning problem-solving concepts by reflecting on problem solving. In F. Bergadano and L. de Raedt, editors, *Machine Learning: ECML-94*, number 784 in Lecture Notes in Artificial Intelligence, pages 287–306. Springer-Verlag, 1994.

[23] Rudi Studer, Henrik Eriksson, John Gennari, Samson Tu, Dieter Fensel, and Mark Musen. Ontologies and the configuration of problem-solving methods. In *Tenth Knowledge Acquisition for Knowledge-Based Systems Workshop*, 1996.

[24] Walter Van de Velde. Towards knowledge level models of learning systems. In *Knowledge level models of machine learning Workshop preprints*, 1994. ML-Net Familiarization workshops, Catania.

[25] F. van Harmelen and J. R. Balder. (ML)2: A formal language for kads models of expertise. *Knowledge Adquisition*, 4(1), 1992. Special Issue 'The KADS approach to knowledge engineering'.

[26] Manuela Veloso and Jaime Carbonell. Toward scaling up machine learning: A case study with derivational analogy in prodigy. In Steven Minton, editor, *Machine Learning Methods for Planning*, pages 233–272. Morgan Kaufmann, 1993.

[27] Bob Wielinga, Walter van de Velde, Guss Schreiber, and H. Akkermans. Towards a unification of knowledge modelling approaches. In J. M. David, J. P. Krivine, and R. Simmons, editors, *Second generation Expert Systems*, pages 299–335. Springer Verlag, 1993.