Reflection in Noos: An object-centered representation language for knowledge modelling

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

1 Introduction

In the development of knowledge-based systems (KBS) an important issue is the degree to which different components can be described, reused and combined. The 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 [Wielinga 93] or components of expertise [Steels 90] are based on the task/method decomposition principle and the analysis of knowledge requirements for methods. Our goal in developing NOOS is to have a language that supports description, reuse, and dynamic combination of components resulting from knowledge modelling analysis in a domain. NOOS is a reflective object-centered representation language that represents *uniformly* problem solving methods and domain knowledge. This uniform representation is possible because of the reflective capabilities of NOOS. Moreover, reflection in NOOS allows a flexible and uniform combination and selection of the different components. Reflection is a powerful principle that allows to organize in a simple and clear way the different types of knowledge involved in KBS design and implementation. Several forms of meta-level reasoning can be described in NOOS in a clear and simple way, for instance implementing a meta-level method that dynamically selects a domain-specific method after analyzing the available information.

The main focus of this paper is to present the reflective capabilities of the NOOS language, so we present first some intuitions about the language and later a formalization of it. An example of using NOOS is also presented, but the reader may be interested in other more detailed applications of NOOS for case-based reasoning (CBR) systems [Arcos and Plaza 93], integrating induction and CBR [Armengol and Plaza 94], and the support NOOS gives for knowledge modelling [Arcos and Plaza 94].

The next section introduces the basic capabilities of NOOS language. In section 3 we will present the formal description of NOOS using Reflective Dynamic Logic (RDL), a logical framework to describe reflective logical architectures [Sierra 95]. Section 4 takes a knowledge modelling analysis of diagnosis tasks performed by R. Benjamin [Benjamins 94] and shows how it can be implemented in NOOS. Finally, section 5 discusses related work and our future work.

2 Basic Notions

The basic elements of the NOOS language are *entities*. Entities represent individuals (or sets of individuals) of the world in a given domain. For instance, in diagnosis of car malfunctions, we define entities that represent cars with malfunctions. The second elements of the NOOS language are *features*. An entity is described by a collection of features. In NOOS there is a functional interpretation of features: a feature of an entity has a functional relation with another entity. This second entity is named the *feature value*.

A first way to define feature values is to refer to other entities. For instance, we can define a `complaint` feature value for `Peters-car` as a reference of `does-not-start` entity (see fig. 1). One feature can refer to one entity or to set of other entities. In general, features are interpreted as functional relations to sets of entities.

Another way to define feature values is to establish a reference with another feature value of some entity. For instance, in our car diagnosis domain the `gas-gauge-reading` feature of `Car` will be defined by a reference to the `gas-level-in-tank` feature. The semantics of this reference

is that gas-gauge-reading is constrained to be the same as gas-level-in-tank (see fig. 1).

```
(define Car
   (gas-gauge-reading (>> gas-level-in-tank ))
   ((empty-level? (define (Identity?)  ; see footnote [1]
                     (item1 empty)
                     (item2 (>> gas-level-in-tank )))))
   (price (>> price model)))

(define (Car Ibiza-Car)
   (model Ibiza))

(define (Ibiza-car Peters-car)
   (owner Peter)
   (complaint does-not-start)
   (gas-level-in-tank full))
```

Figure 1. Definition of the entity car and definition by refinement of two new entities: Ibiza-car and Peters-Car.

There are two ways to define references: absolute references and relative references. An absolute reference is (>> feature of entity) where feature and entity refer to the name of some feature and the name of an entity. A relative reference is a reference where the entity reference is omitted (>> feature), in this case the entity implicitly referred to is the *root* entity in the lexical scope of the definition. In Fig. 1 the roots are car, Ibiza-car and Peters-car.

We can also establish a reference with another feature of some entity indirectly by means of intermediate feature references. This composition of features is called *path*. For instance, we can establish the price of a given car as the price of the model of this car, writing (>> price model) in the price feature (see fig. 1).

We can use a more complex way to describe a feature values: we can define a *method*. A method can be understood as a function with a set of parameters. We will explain methods in detail below. In the previous example we use the built-in identity? method to describe the empty-level? feature of car. The empty-level? feature value will be true when gas-level-in-tank is empty and false otherwise (identity? method works like eq predicate of Lisp).

Another way of defining entities is by refinement. A new entity is refined from another entity by adding more features or redefining existing ones. For instance, we can define a car entity with the common knowledge about cars, then defining models of cars by refinement, and finally define concrete cars with the specific information of each car by refinement from models of cars (see fig. 1).

Methods are also entities, but they are evaluable entities. Features in a method are viewed as *tasks*. Specifically, the set of features defined in a method 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. For instance, a generate-and-test method will be decomposed into the generate and test subtasks. This recursive decomposition of task into subtasks by means of a method is called the task/method decomposition.

The NOOS language provides a set of basic built-in methods, and we can define new methods from this set of basic methods (definition by refinement). Examples of NOOS built-in methods are arithmetic operations, conditional, set operations, logic operations and operations for comparing entities. For instance, we define a causal-explanation method as a conditional-method where cause is equivalent to condition and effect is equivalent to result. Refining causal-explanation we can define specific causal explanations, like c1 below. In c1 a battery-voltage of a car equal to low-voltage justifies the conclusion of having a low-battery-malfunction.

```
(Define (Causal-explanation C1)
   (car )
   ((cause (define (Identity?)
              (item1 low-voltage)
              (item2 (<< battery-voltage car)))))
   (effect low-battery-malfunction))
```

Another example of definition of a method is the definition of a generate and test method for the diagnosis of car malfunctions. This method is defined from a built-in method named decomposition-method. Decomposition-method allows the definition of a sequential chaining of subtasks (relative to the writing order) and returns the result value of the last subtask.

```
(Define (Decomposition-method Generate&Test)
   ((generate-hypothesis specific-generate-method-1))
   ((test-hypothesis specific-test-method-2)))
```

where these methods are defined elsewhere. Section 4 shows a more detailed generate and test method for diagnosis.

In complex domain problems usually there are alternative ways to define methods for a feature value. This happens because sometimes we can infer the value of a feature using different knowledge in each method and we don't know in advance which information will be available. In this case, we can define a *metalevel entity* that contains these alternative methods.

In the following example the meta of Peters-car is defined with methods for Peters-car.

[1] Noos syntax to define a method for a feature is with a double parenthesis

```
(Define (Meta of Peters-Car)
    ; these are methods for empty-level? features
    ; of Peters-car
    (empty-level? gas-gauge-reading-explanation
                  gas-level-in-tank-explanation)
    ; this is a metalevel method that computes
    ; methods for diagnosis
    ((diagnosis
      (define (select-car-explanations-method)
       (current-complain (>> complaint of (referent)))))
)   ; see footnote[2]
```

A metalevel entity is just an entity plus a metalevel relation with a (base-level) entity (called *referent* entity of the metalevel entity). The features defined in a metalevel entity has a corresponding feature in the base level with the same name. Since metalevel entities are defined in the same way that other entities we can define feature values of metalevel features referencing other entities, defining a path or defining a (metalevel) method.

The definition of feature values by means of references allows to define directly a set of applicable methods for a given feature. NOOS allows to enrich this description of methods adding a partial order among the set of methods. We call this sets *posets* (partially ordered set). In this case the language semantics guarantees that this partial order will be interpreted as a preference ordering for finding the most preferable feature value for the feature of the referent entity. This process will be explained in section 2.1. For instance, defining a specific to general ordering among methods forces to the system to use first more specific methods and, if these fail, use then the more general ones. A metalevel feature value can be defined with a path, this allows a metalevel entity to refer to some methods described in another entity. The last way of defining a metalevel feature value is by means of a (metalevel) method[3]. This metalevel method will allow to obtain a set of plausible useful methods for that feature. These methods can be selected by the metalevel method taking into account the given information of the current problem. We have shown elsewhere that case-based reasoning methods [Arcos and Plaza 93] and inheritance [Plaza 92] can be defined as metalevel methods. Another example is using a generate and test strategy for selecting (from a set of possible hypotheses) causal explanation methods according to the current complaint and then test which of them fits in the current problem [Arcos and Plaza 94].

NOOS language is uniform: all methods and metalevel entities are also entities. Therefore, we can define a metalevel entity of any entity, including one that is metalevel entity with respect to a third entity, etc. This possibility can be useful, for instance, when we have several ways (metalevel methods) to obtain methods to solve a task.

Another important remark is that this uniform representation of methods as entities with a set of features interpreted as subtasks allows to define multiple methods to achieve a subtask by means of the metalevel feature description of the subtask. For instance, in the `generate` subtask of a `generate-and-test` method we can define a set of multiple ways to generate plausible hypotheses to be tested. The `test` subtask may also have several methods.

```
(Define (Meta of Generate&Test)
    (generate-hypothesis generator-method1
                         generator-method2)
    (test-hypothesis test-method1 test-method2))
```

2.1 Noos Queries

Up to now we have seen what we can describe in NOOS, but which type of inference can NOOS perform ? The inference process in NOOS starts by means of *queries*. Queries are demands about feature values of entities. When a query is asked to the system a task is engaged. This task forces to infer the feature value of the entity. If there is a method or path specified for this given feature, the new task of the system is to evaluate this method or path. Moreover, this evaluation may engage subtasks that need to be solved and these subtasks will force recursively the evaluation of other methods.

When there is no specified method in the base level entity for a given feature an *impasse* occurs and the control of inference is passed to the metalevel entity. The metalevel attempts to infer a partially ordered set of methods. Then, preserving this partial order among methods, the preferred method is reflected down at the base level and the control is returned to the base level. If the base level with this method is not capable to obtain a feature value then a new impasse occurs and the next preferred method is reflected down. This inference cycle is repeated until one method succeeds or the metalevel is not capable to obtain more methods to reflect down. This approach involves backtraking as a constitutive aspect of NOOS, as can be expected for plausible reasoning applications[4].

---

2 Referent allows to refer to the object entitiy bounded with a given metalevel entity. In this case, referent of `meta` of `Janets-car` refers to `Janets-car`.

3 Therefore, all feature value definitions at the metalevel are exactly like those we specified for the base level

4 Case-based reasoning methods are implemented in Noos by means of metalevel methods that retrieve methods for a feature (e.g. diagnosis) from other "similar" entities

The queries given in NOOS are *infer-value*, *known-value*, *defined-value* and *all-values*. *Known-value* is a query of a feature value for a given entity that returns that value only if it is already known; if deduction is needed to infer the value it returns undecided. *Infer-value* is a query of a feature value for a given entity with deduction. *Defined-value* is a query that determines if it is possible to deduce any defined feature value. The answer of this query will be `true` if it is possible and `false` if the value is undefined. *All-values* is a query that determines all the deducible values for a given feature of an entity.

3 Noos Formalization

We will describe our system using Reflective Dynamic Logic [Sierra 95]. Reflective Dynamic Logic (RDL) is a propositional dynamic logic [Harel 84] to describe architectures to build reflective knowledge-base systems with complex reasoning patterns.

3.1 Reflective Dynamic Logic

In general, a reflective architecture allows to build reflective knowledge-bases (RKB) as a set of *units* with initial local theories in possibly different languages. Each unit is also allowed to have its own intra-unit deductive system. Moreover, the whole RKB is equipped with an additional set of inference rules, called reflection rules, to specify the information flow among the different units of the RKB. For a full description of RDL see [Sierra 95]. Here we only present some basic definitions of RDL that we use later:

Definition 1 *Reflective Knowledge-Based System* is $RKB = (U, R)$, where $U = \{u_i\}_{i \in I}$ is a set of units written in some language, concretely $u_i = (L_i, \Omega_i^0, \Delta_i)$, where L_i is the language, Ω_i^0 a set of formulas written in that language ($\Omega_i^0 \subset \{\phi \mid \phi \in L_i\}$), Δ_i is a set of inference rules and $R = \bigcup_{i,j \in I} R_{ij}$ is a set of reflection rules between units.

Definition 2 Given a RKB, the set of atomic formulas of RDL is defined as

$$\Phi_0 = \{ i : \varphi \mid i \in I, \varphi \in L_i \}$$

where $i \in I$ denotes the index of a unit, that is, formulas are indexed by unit names, using the notation *unit_index/formula*.

Definition 3 Given a RKB, the set $\prod_0$ of *atomic programs* of RDL is defined as the union of the intra-unit inference rules $\prod_0^{Intra}$ and the RKB reflection rules $\prod_0^{Inter}$.

$$\prod_0 = \prod_0^{Intra} \cup \prod_0^{Inter}$$

$$\text{where } \prod_0^{Intra} = \bigcup_{i \in I} \Delta_i \text{ and } \prod_0^{Inter} = \bigcup_{i,j \in I} R_{ij}$$

Definition 4 Given the above set of atomic programs we define two of sets of undeterministic compound programs as

- Intra-unit deduction is $\vdash_k = \bigcup_{\alpha \in \prod_{0_k}^{Intra}} \alpha$
- Inter-unit deduction is $\vdash_{kl} = \bigcup_{\alpha \in \prod_{0_{kl}}^{Inter}} \alpha$

3.2 Noos formal syntax

Formally, NOOS entities are described by units in RDL. A language L of a NOOS unit is defined by the signature

$$\Sigma = \Sigma_e \cup \Sigma_f \cup \Sigma_b \cup \{\bot, \varnothing, true, false\}$$

where Σ_e is a set of symbols of entities, Σ_f a set of symbols of features, Σ_b a set of symbols of built-in functions, and $\{\bot, \varnothing, true, false\}$ are undefined, empty set, true and false symbols respectively.

The terms in language L are formed according to the rules:

Elementary terms: Elementary terms in NOOS are entities, undefined, empty set, true, false, sets of entities and partially ordered sets of entities.

$$\Sigma_e \subset T_0,$$
$$\bot, \varnothing, true, false \in T_0,$$
$$e_1, \ldots, e_i \in T_0: \quad \{e_1 \cdots e_i\} \in T_0$$
$$e_1, \ldots, e_i \in T_0, \ \prec \text{ is a partial order defined on } \{e_1 \cdots e_i\}: \quad [e_1 \cdots e_i | \prec] \in T_0$$

Paths: There are four ways of construct paths according the four types of queries allowed in NOOS, namely infer-value (.), known-value (!), defined-value (?), and all-values (@).

$$T_0 \subset T_1,$$
$$t \in T_1,\ f \in \Sigma_f: \ t.f,\ t!f,\ t?f,\ t@f \in T_1$$

Built-in functions: Finally we can define methods as terms with a set of named subtasks.

$$T_1 \subset T,$$
$$f_1, \ldots, f_n \in \Sigma_f,\ t_1, \ldots, t_n \in T,\ b \in \Sigma_b: \quad b(\langle f_1 | t_1 \rangle \cdots \langle f_n | t_n \rangle) \in T,\ \forall n \in \mathbb{N}$$

("cases" or previously solved problems). Some CBR methods rank the retrieved methods in a preference order based on the degree of similarity between the current problem and precedent cases. [Arcos and Plaza 93]

Formulas of the language are:

form1 : The first type of formulas allows to express feature values of entities. A feature value can be defined as an entity, a set of entities, a path or a method.

$$f \in \Sigma_f,\ e \in \Sigma_e,\ t \in T:\ e.f \doteq t \in \Phi$$

form2 : The second type of formulas allows to express the set of possible feature values. This type of formulas are used by the all-values query.

$$f \in \Sigma_f,\ e \in \Sigma_e,\ t \in T_0:\ e@f \doteq t \in \Phi$$

There are two notational equivalences that will simplify the definition of the NOOS inference rules:

Definition 5. Every entity is considered equivalent to the singleton set that contains this entity.

$$\forall e \in \Sigma_e \quad e \equiv \{e\}$$

Definition 6. A set of entities is considered equivalent to a partially ordered set with the same entities and no ordering among them.

$$\forall \{c_1 \cdots c_j\} \subset T_0 \quad \{c_1 \cdots c_j\} \equiv [c_1 \cdots c_j | \prec_\varnothing]$$

3.3 Inference Rules

Before describing the NOOS inference rules some notational conventions are given below:

i) $t|_p = X$ means that there is a part p of the term t that contains the X subterm.

ii) $t[X']_p$ means that a new term is constructed replacing the part p in t by X'.

iii) c symbols refer to entities, f symbols refer to features and s symbols refer indistinctly to an entity or a set of entities.

3.3.1 Intra-unit inference rules

The first set of intra-unit inference rules is related to the path reduction. For instance, inference rule δ_1 describes the (simple) path reduction step

$$\frac{\begin{array}{c} c.f \doteq t \\ t|_p = c_1.f_k \\ c_1.f_k \doteq s_1 \end{array}}{c.f \doteq t[s_1]_p} \qquad [\delta_1]$$

In general, (usually for features at metalevel entities) feature values are posets. The inference rule δ_2 describes the general path reduction step

$$\frac{\begin{array}{c} c.f \doteq t \\ t|_p = [c_1 \cdots c_n | \prec].f_k \\ c_1.f_k \doteq s_1 \\ \cdots \\ c_n.f_k \doteq s_n \end{array}}{c.f \doteq t\left[\{s_1 \cdots s_n | tr(\prec, f_k)\}\right]_p} \qquad [\delta_2]$$

where tr is a function that transforms the given partial order to a new partial order over the new set:

$$tr(\prec, f_k) = \left\{ s_i < s_j \;\middle|\; \begin{array}{l} c_i < c_j \in \prec \ \&\ c_i.f_k \doteq s_i \\ \&\, c_j.f_k \doteq s_j \ \&\ i \neq j \end{array} \right\} / \left\{ s_i < s_j \,\middle|\, s_j <_* s_i \right\}$$

There is also a set of intra-unit inference rules describing the inference step application of each built-in method provided by NOOS. For instance, rule a_1 express the usual interpretation of the numeric addition operation:

$$\frac{\begin{array}{c} c.f \doteq t \\ t|_p = add\left(\langle items | \{c_1, \cdots, c_n\} \rangle\right) \\ \text{“}c' = c_1 + \cdots + c_n\text{”} \end{array}}{c.f \doteq t[c']_p} \qquad [a_1]$$

3.3.2 Inter-unit inference rules

There are three kinds of inter-unit inference rules: *Reification rules* specify the representation that a metalevel unit has about its corresponding base level unit. *Reflection rules* specify the changes that a metalevel unit may perform upon its corresponding base level unit. Finally the *translation rules* specify how formulas may be transported from a unit to another one. In the following we will use Kleene quotes to mean the quoting $\ulcorner t \urcorner$ and unquoting $\llcorner t \lrcorner$ of a term t.

The reification rule $R^{up}_{o_i m_i}$ adds in metalevel entity m_i the set formulas about the feature values known in the entity o_i

$$R^{up}_{o_i m_i} = \left\{ \frac{c.f \doteq c'}{\ulcorner c.f \urcorner.referent \doteq c'} \right\} \quad \text{where } \ulcorner c.f \urcorner \in \Sigma_{m_i}$$

The reflection rule transforms a method entity to a method description in the base-level language and installs this description to the same feature of the referent.

$$R^{down}_{o_i m_i} = \left\{ \frac{c_{m_i}.f \doteq [c_1 \cdots c_n | \prec]}{c_{o_i}.f \doteq \llcorner c_j \lrcorner} \right\}$$

where $\llcorner c_j \lrcorner$ is defined as

$$c_j.t_1 \doteq T_1$$
$$\cdots$$
$$\frac{c_j.t_n \doteq T_n}{c_j\left(\left\langle t_1|T_1\right\rangle\cdots\left\langle t_n|T_n\right\rangle\right)}$$

The translation rules just obtain feature value formulas from other entities. For instance,

$$R^{query}_{o_i f_k o_j} = \left\{\frac{c_{o_i}.f_k \doteq c'}{c_{o_i}.f_k \doteq c'}\right\}$$

Notice that renaming of imported formulas is not necessary because NOOS formulas already contain the information of which is the unit of origin (c_{o_i}).

3.4 Noos Programs

NOOS queries are formalized in RDL as programs. There are four types of queries in NOOS: infer-value, known-value, defined-value and all-values. Infer-value can be defined as a program $\pi^{iv}_{c_j f_k c_i}$ (meaning that c_i asks the feature value of f_k to entity c_j) as follows:

$$\pi^{iv}_{c_j f_k c_i} = \pi_{c_j f_k}; R^{query}_{c_j f_k c_i} \cup$$
$$R^{up\ t}_{c_j m_j}; \pi^{iv}_{m_j f_k}; R^{down}_{m_j c_j}; \pi_{c_j f_k}; R^{query}_{c_j f_k c_i}$$

Notice that Infer-value triggers a query to the metalevel (where m_j is the metalevel of c_j) when the base level c_j is unable to answer the query.

Known-value query is a program where no inference is engaged:

$$\pi^{known}_{c_j f_k c_i} = R^{query}_{c_j f_k c_i}$$

Defined-value will be defined as

$$\pi^{def}_{c_j f_k c_i} = \pi^{iv}_{c_j f_k c_i} \cup \neg\left(\left\langle \pi^{iv}_{c_j f_k c_i}\right\rangle\right)?; R^{def}_{c_j f_k c_i}$$

Finally, the all-values query is defined as performing inference until no more new values (formulas) can be deduced:

$$\pi^{av}_{c_j f_k c_i} = \left\langle \pi^{next\ *}_{c_j f_k c_i}\right\rangle \bigwedge_{\varphi\in L_{c_i}}\left(\left\langle \pi^{next}_{c_j f_k c_i}\right\rangle\varphi \to \varphi\right)?$$

$$\pi^{next}_{c_j f_k c_i} = \pi^{iv}_{c_j f_k c_i}; R^{av}_{c_j f_k c_i}$$

The local inference program $\pi_{c_j f_k}$ is the program engaged by c_j when it receives a query for feature f_k. It starts the internal inference in entity c_j and engages all the necessary (sub)queries to other entities (and to itself) in order to find a feature value for f_k. The local inference program is defined as:

$$\pi_{c_i f} = (\ \vdash^t_{c_i}\ \cup$$
$$\langle\vdash^t_{c_i}\rangle \bigvee_{\varphi\in L_{c_i}} c_i.f \doteq \varphi \wedge \varphi|_p = \left[c_j\cdots c_l|\prec\right].f_k\,?;$$
$$\pi^{iv}_{c_j f_k c_i};\cdots;\pi^{iv}_{c_l f_k c_i} \cup$$
$$\langle\vdash^t_{c_i}\rangle \bigvee_{\varphi\in L_{c_i}} c_i.f \doteq \varphi \wedge \varphi|_p = \left[c_j\cdots c_l|\prec\right]!f_k\,?;$$
$$\pi^{known}_{c_j f_k c_i};\cdots;\pi^{known}_{c_l f_k c_i} \cup$$
$$\langle\vdash^t_{c_i}\rangle \bigvee_{\varphi\in L_{c_i}} c_i.f \doteq \varphi \wedge \varphi|_p = \left[c_j\cdots c_l|\prec\right]@\,f_k\,?;$$
$$\pi^{av}_{c_j f_k c_i};\cdots;\pi^{av}_{c_l f_k c_i} \cup$$
$$\langle\vdash^t_{c_i}\rangle \bigvee_{\varphi\in L_{c_i}} c_i.f \doteq \varphi \wedge \varphi|_p = c_j?f_k\,?;\pi^{def}_{c_j f_k c_i} \cup$$
$$)^*; \bigvee_{s\in T_0} c_i.f \doteq s\,?$$

4 An example for diagnosis tasks

We will use as example the knowledge modelling analysis of diagnosis tasks performed by R. Benjamin [Benjamins 94]. The original analysis was intended for the KADS knowledge modelling framework [Wielinga et al, 92] and we intend to show here how NOOS offers a computational support using reflection to implement these ideas developed for knowledge acquisition. The general scheme for diagnosis task is showed in figure 2.

We can now refine this description by a method that is composed of two tasks, namely `Generate-Hypothesis` and `Discriminate-Hypothesis` (see figure 3).

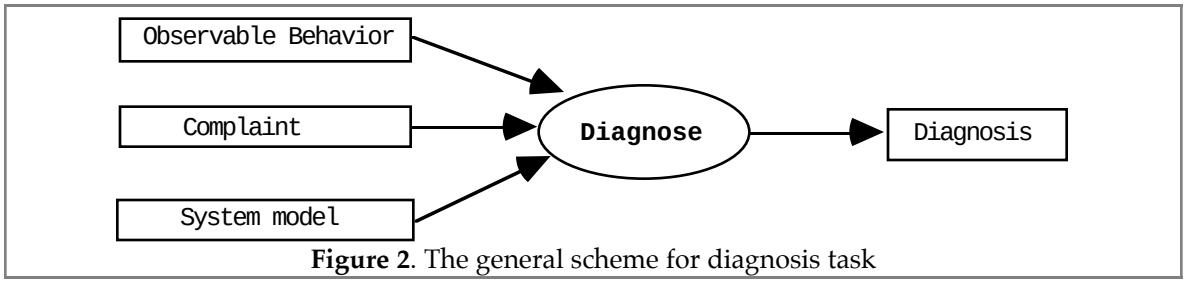

Figure 2. The general scheme for diagnosis task

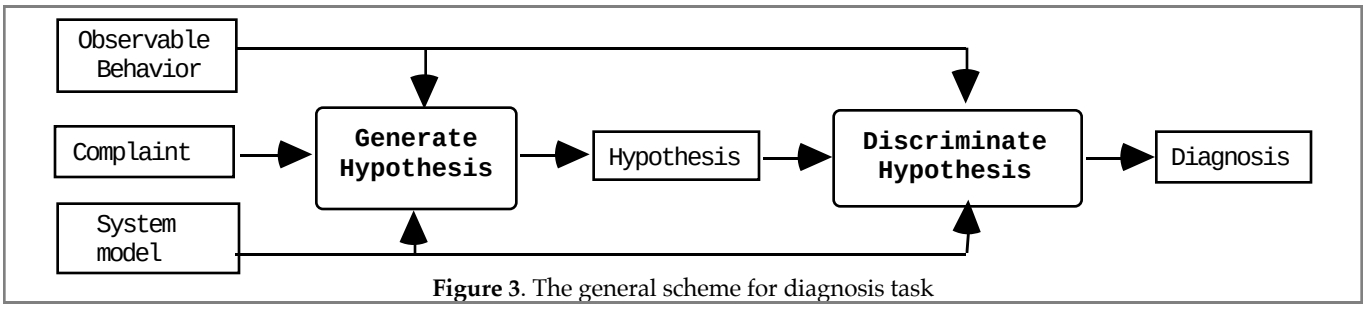

Figure 3. The general scheme for diagnosis task

The specific methods that can be used to achieve those tasks `Generate-Hypothesis` and `Discriminate-Hypothesis` are very varied, and essentially they are different because they use the knowledge of the different kind of models we may have of a system, like behavior models, associations models and causal dependency models. NOOS allows to include for every task all available methods: as shown in the figure below (fig. 5), where the General Diagnosis is a method that has three tasks: `Detect-Complaint`, `Generate-Hypothesis`, and `Discriminate-Hypothesis`. Notice, for instance, that the `Generate-Hypothesis` task has two methods, namely `Model-based-hypothesis-generation` and `empirical-hypothesis-generation`. Notice also that each method is a NOOS entity that defines its specific subtasks (e.g. for the first method tasks are `find-contributors`, `transform-to-hypothesis-set` and `transform-to-hypothesis-set`); each of those subtasks may also have multiple alternative methods (e.g. `find-contributors` subtask has `trace-back`, `causal-covering`, and `prediction` as methods that may achieve that task).

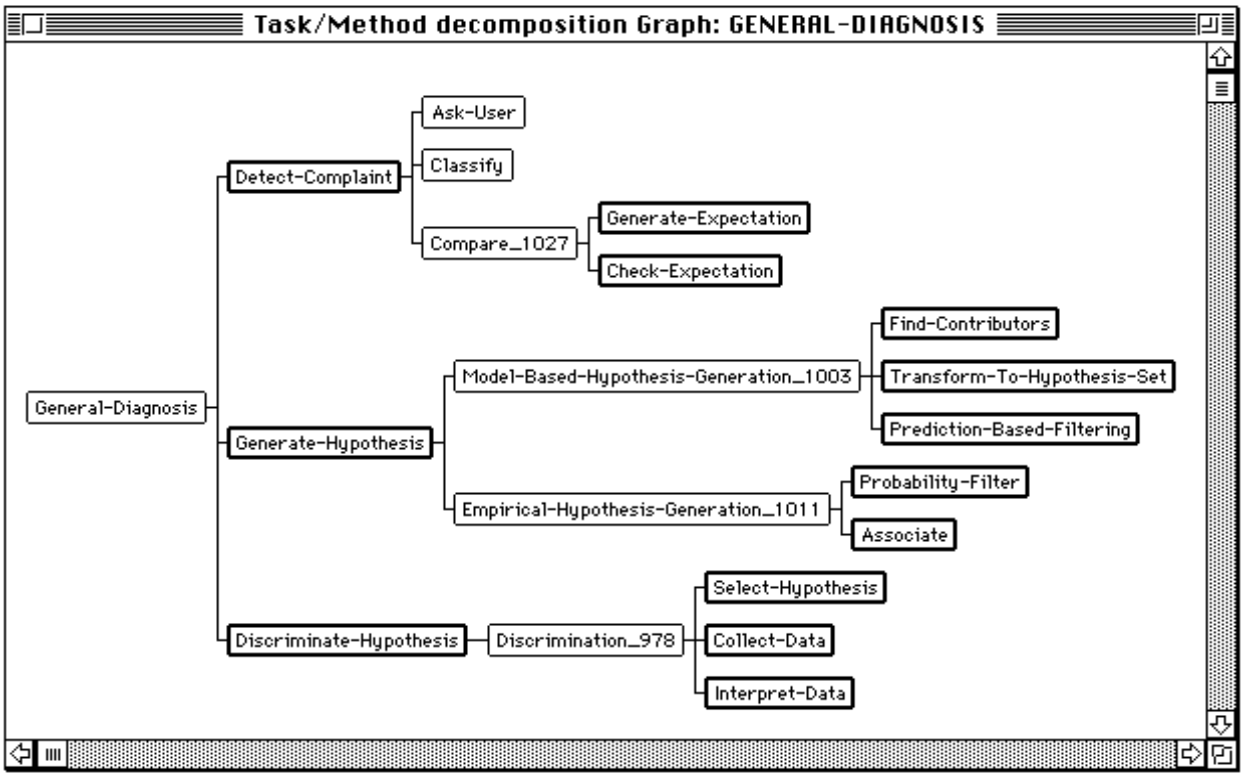

Figure 4. A browser showing part of the task/method decomposition for method `general-diagnosis`.

```
(define (Decomposition-Method General-Diagnosis)
  ((detect-complaint ask-user
                     classify
                     (define (Compare)
                         ((generate-expectation look-up
                                                simulate))
                        ((check-expectation threshold-method
                                            constraint-check)))))
  ((generate-hypothesis
    (define (Model-based-hypothesis-generation)
      ((find-contributors trace-back
                          causal-covering
                          prediction))
      ((transform-to-hypothesis-set set-cover
                                    intersection
                                    subset-minimality
                                    cardinaslity-minimality))
      ((transform-to-hypothesis-set constraint-suspension
                                    corroboration
                                    fault-simulation)))
    (define (empirical-hypothesis-generation)
      ((associate associate-method))
      ((probability-filter prob-filter-method)))))
  ((discriminate-hypothesis
    (define (discrimination)
      ((select-hypothesis random
                          (define (smart)
                              ((estimate-cost-hypothesis-set local-cost-estimate
                                                             number-of-tests-estimate
                                                             overall-costs-estimate))
                             ((order-hypothesis-set ordering-method))
                            ((select-first take-first)))))
      ((collect-data  compiled-test
                     (define (probing)
                        (obtain)
                         (generate-expectation)
                        (compare))
                     (define (manipulating)
                         (reduce-input-vector)
                        (simulate)
                        (obtain)
                        (compare))
                     (define (replace)
                         (replace-hypothesis)
                         (generate-expectation)
                        (compare))))
      ((interpret-data interpret-in-isolation
                       split-half-interpret
                       model-based-hypothesis-generation))))))
```

Figure 5. A complex example of multiple methods for generic diagnosis tasks. Every method determine its own subtasks and exploit different knowledge models of the system being diagnosed. The code shown is simplified because some parts are not included for readability reasons.

Another capability of NOOS, not shown here is that there can be a meta-level method for each task that can choose the appropriate for some problem dynamically, for instance checking whether the problem has some information required for the available methods, having some information of the probability of success of a method in *similar* problems (case-based reasoning and learning has been implemented in NOOS, as shown in [Arcos and Plaza 94]), etc.

5 Related work and Conclusions

Related work on reflection is [Kiczales 91], [Giunchilia 90], and [Smith 85]. Meta-level architectures have been used for strategic reasoning [Godo 89] [Lopez 93], for non-monotonic reasoning [Treur 91], and for modelling expert systems

[Akkermans 93]. Precedents on using reflection for case-based reasoning are [Ram 92; López 93]. Our work on architectures is related to cognitive architectures like SOAR [Newell 90], THEO [Mitchell 91]. However, THEO does not provide a clear metalevel definition. Related work on knowledge-level modelling of AI systems includes the COMMET (or components of expertise) framework [Steels 90], and the KADS methodology [Akkermans 93]. KADS has reflective framework, called "knowledge-level reflection" that uses the KADS model to specify the system self-model of structure and process, very much like our inference-level model of entities, tasks, and methods.

The NOOS design was guided by the goal of following knowledge modelling analysis in KBS construction and the integration of learning methods in the same framework. We performed a knowledge modelling analysis of case-based reasoning and learning [Armengol and Plaza 93] and we have used NOOS to implement CHROMA, a KBS for purification of proteins from animal tissues. CHROMA learns from experience using two learning methods: CBR learning and induction. The reflective capabilities of NOOS allow to analyse and decompose problem solving and learning methods in a uniform way and to combine them in a simple and efficient way. Another classification-based system for sponge identification is being developed at our Institute for a class of marine sponge species. Our purpose is also using NOOS to integrate several problem solving and learning methods for this task.

The design decisions that have shaped NOOS arise from one basic intuition: reflection is clean and powerful mechanism to represent different types of knowledge in separate layers and then define the relationship among them. Previously, several AI systems developed at our Institute, MILORD [Godo 89] and BOLERO [López 93] used the distinction between metalevel and base level embody the difference between domain reasoning and strategic reasoning. The capabilities of NOOS allow us first to analyze a task domain, describing its structure using knowledge modelling, and then construct an appropriate KBS architecture. The example of section 4 shows an architecture for diagnosis tasks based on such an analysis.

Acknowledgements

The research reported on this paper has been developed at the IIIA inside the ANALOG Project funded by CICYT grant 122/93 and a CSIC fellowship.

References

Akkermans, H., van Harmelen, F., Schreiber, G., Wielinga, B (1993) A formalisation of knowledge-level model for knowledge acquisition. *Int Journal of Intelligent Systems* , 8(2):169-208.

Armengol, E., Plaza, E.: Integrating induction in a case-based reasoner. Proc. 2nd European Workshop on Case-based Reasoning, (to appear).

Arcos, J. L., and Plaza, E. (1993), A Reflective Architecture for Integrated Memory-based Learning and Reasoning, In S. Wess, K.D. Althoff, M.M. Richter (Eds.), *Topics in Case-Based Reasoning. Lecture Notes in Artificial Intelligence*, 837, p. 289-300. Springer Verlag: Berlin.

Arcos, J.L., Plaza, E. (1994), Integration of Learning into a Knowledge modelling framework. *Lecture Notes in Artificial Intelligence*, Vol. 867. Springer-Verlag 1994, pp. 355-373. Available online at URL "`http://www.iiia.csic.es/People/enric/EKAW-94_ToC.html`"

Benjamins, R., (1993): "Problem Solving Methods for Diagnosis". PhD thesis, University of Amsterdam.

Benjamins, R., (1994): "On a Role of Problem Solving Methods in Knowledge Acquisition - Experiments with Diagnostic Strategies-". Proceedings EKAW-94. *Lecture Notes in Artificial Intelligence*, Vol. 867. Springer-Verlag.

Giunchilia, F., and Traverso, P., (1990), Plan formation and execution in an architeture of declarative metatheories. *Proc of META-90: 2nd Workshop of Metaprogramming in Logic Programming..* MIT Press.

Godo, L, López de Màntaras, R, Sierra, C, Verdaguer, A (1989), MILORD: The architecture and the management of linguistically expressed uncertainty. *Int. J. Intelligent Systems*, 4:471-501.

Harel, D. (1984): "Dynamic Logic", in D.M. Gabbay and F.Guenthner eds. *Handbook of Phylosophical Logic* vol2, Kluwer, p. 497-604.

Kiczales G, Des Rivières J, Bobrow D G. *The Art of the Metaobject Protocol*, The MIT Press: Cambridge, 1991.

López, B, and Plaza, E, (1993), Case-based planning for medical diagnosis, In Z. Ras (Ed.) *Methodologies for Intelligent Systems*, p. 96-105. Lecture Notes in Artificial Intelligence: Springer-Verlag

Mitchell, T.M., Allen, J., Chalasani, P., Cheng, J., Etzioni, O., Ringuette, M., Schlimmer, J. C. Theo: a framework for self-improving systems. In K Van

Lenhn (Ed.) *Architectures for Intelligence*. Laurence Erlbaum, 1991.

Newell, A (1990), *Unified Theories of Cognition*. Cambridge MA: Harvard UniversityPress

Plaza, E (1992), Reflection for analogy: Inference-level reflection in an architetcure for analogical reasoning. *Proc. IMSA'92 Workshop on Reflection and Metalevel Architectures*, Tokyo, November 1992, p. 166-171.

Ram, A, Cox, M T, Narayanan, S. (1992), An architecture for integrated introspective learning. *Proc. ML'92 Workshop on Computational Architectures for Machine Learning and Knowledge Acquisition*.

Sierra, C., Godó, Ll., López de Mántaras, R. (1995): "A dynamic logic framework for reflective Architectures". Submited the IJCAI-95 Workshop on Reflection and Metalevel Architecture and their applications in AI.

Smith, B.C. Reflection and semantics in a procedural language, In Brachman, R. J., and Levesque, H. J. (Eds.) *Readings in Knowledge Representation*. Morgan Kauffman, California, 1985, pp. 31-40.

Steels, L. (1990) The Components of Expertise, *AI Magazine*, **11**(2):29-49.

Treur, J (1991), On the use of reflection principles in modelling complex reasoning. *Int. J. Intelligent Systems* , 6:277-294.

Wielinga, B, Schreiber, A, Breuker, J (1992), KADS: A modelling approach to knowledge engineering. *Knowledge Acquisition* **4**(1).