To appear in *Future Generation Computer Systems* (1996)

Inference and reflection in the object-centered representation language Noos

Josep Lluís Arcos Enric Plaza

Artificial Intelligence Research Institute, IIIA
Spanish Council for Scientific Research, CSIC
Campus UAB,
08193 Bellaterra, Catalonia, Spain.
{arcos,enric}@iiia.csic.es

Abstract

This paper explains the inference and reflection capabilities of NOOS, an object-centered representation language designed to integrate problem solving and learning. Problem solving and learning in NOOS are modelled by means of concepts, tasks, methods and metalevels. Metalevels allow NOOS to reason *about* own problem solving. Using metalevels, NOOS can reason about preferences in order to make decisions about sets of alternatives present in domain knowledge and problem solving knowledge. Reflection in NOOS is provided by inference processes that involve metalevels. Basic reflective capabilities include reasoning about alternative methods to solve a task, reasoning about what is known by the system itself, and reasoning about the existence of solutions. A formal model of NOOS inference using Descriptive Dynamic Logic is also presented.

1 Introduction

In the development of knowledge-based systems (KBS) an important issue is the degree to which different knowledge 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 [20] or components of expertise [17] 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 domain knowledge, problem solving methods, and learning methods. 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 metalevel reasoning can be performed in NOOS in a clear and simple way, for instance implementing a metalevel 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 notions about the language and later a formalization. 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 [2], integrating induction and CBR [6], and the support NOOS gives for knowledge modelling [3].

The next section introduces the basic capabilities of NOOS language. In section 3 we will present a formal description of the inference process in NOOS using Descriptive Dynamic Logic (DDL), a logical framework to describe reflective architectures [16]. Section 4 takes a knowledge modelling analysis of diagnosis tasks and shows how it can be implemented in NOOS. Finally, section 5 discusses related work and our future work.

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.

In this section we will present the reflective object-centered language NOOS. First we will present the knowledge categories of our model in order to show the framework and motivations of design decisions in the NOOS language. Then we will describe the NOOS language and how the components of model are mapped to the NOOS language, and finally we will explain the inference process in NOOS.

2.1 The NOOS Model

The first category in our model is *domain knowledge*. The domain knowledge category specifies a set of *concepts* and a set of *relations* among them relevants 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 individuals like cars or abstractions like malfunctions. An example of a relation from cars to persons is the *"owner of a car"*.

Another category in our model 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 domain is to diagnose car malfunctions. In our approach *methods* model the ways to solve problems. Methods can be elementary or can be decomposed in 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 the task). For instance, a `generate-and-test` method is decomposed into the `generate` and `test` subtasks and there are several possible methods to achieve each subtask (see section 4). This recursive decomposition of task into subtasks by means of a method is called the task/method decomposition. 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 in our model 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*. 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 models criteria for preferring some methods over other methods for a task in a specific situation. An example of metalevel task is to choose 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 sorts them using a set of preferences.

2.2 The NOOS Language

NOOS is an object-centered representation language where the basic elements are *entity descriptions* embodying a collection of *feature descriptions* characterizing that entity. All the components

```
(define Car
  (owner (define (person)))
  (gas-level-in-tank level)
  (gas-gauge-reading (>> gas-level-in-tank))
  ((empty-level?  (define (Identity?)
                    (item1 empty)
                    (item2 (>> gas-level-in-tank)))))
  (model car-model)
  (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`. For brevity, the definitions of some entities like `car-model` or `full` are not included. Syntax is summarized in figure 2

of the NOOS model are mapped into the language as entity or feature descriptions. This means that with a small set of computational elements we capture all the elements of the model.

The basic elements of the NOOS language are *entity descriptions*. Entity descriptions have three parts: *constituent*, *name* and *body*. The entity *name* is a symbol that denotes the entity; the name is optional and when it is not given an anonymous entity is described. The *body* is a set of feature descriptions related to the entity. *Feature descriptions* are pairs of feature names and feature values. The *constituent* symbol, that is optional, indicates that the entity is a refinement of the entity denoted by this constituent symbol. Concepts, as defined by the NOOS model, are mapped to the NOOS language as *entity descriptions*. For example, concepts like cars or malfunctions are mapped to entity descriptions (see Figure 1).

The notion of refinement is introduced as a methodology to define entity descriptions in the NOOS language. The main idea behind the refinement is that several concepts of the model share many features and tasks. Usually, some concepts are specializations of other more general concepts. A new entity description defined as a refinement of another entity description includes all the feature descriptions defined in the constituent body not redefined in the new body. For instance, the `car` entity can be defined with the common knowledge about cars (figure 1 shows this example and figure 2 describes the NOOS syntax). Then specific models of cars can be defined by refinements of car, and including specific information of each car by refinement from models of cars. All the features defined in `car` and `Ibiza-car` not redefined in `Peters-car` are included in `Peters-car` entity description.

Relations are mapped to feature descriptions. Specifically, the body of a given entity description defines the set of features related to this entity. For instance, in Figure 1 `owner`, `gas-gauge-reading`, `empty-level?` and `price` features are defined in the `car` entity description, `model` feature is defined in `Ibiza-car` and finally `owner`, `complaint` and `gas-level-in-tank` features are defined in `Peters-car` description.

There are three ways to define feature values in an entity description. A first way to define a feature value is to refer to other entities by their name. For instance, a feature value for the `complaint` feature in `Peters-car` can be defined as a reference to `does-not-start` entity (see Figure 1). A feature can refer to an entity or to a set of entities. Thus, features are interpreted

entity-description	::=	named-description
	|	named-ref-description
	|	metalevel-description
named-description	::=	**(define** *entity-name* feature-description*)
named-ref-description	::=	**(define** (*constituent entity-name*) feature-description*)
anonymous-description	::=	**(define** (*constituent*) feature-description*)
metalevel-description	::=	**(define** (*constituent* **(meta+ of** *entity-name*)) feature-description*)
feature-description	::=	(*feature-name entity-ref+*)
	|	(*feature-name* query-description)
	|	((*feature-name* entity-ref))
entity-ref	::=	*entity-name* | anonymous-description
query-description	::=	(op *feature-name** [**of** *entity-name*])
op	::=	`>>` | `?>>` | `!>>` | `*>>`

Figure 2: This figure shows a subset of NOOS syntax used for the definition of entity descriptions in BNF notation. Remark that in feature-description double parenthesis are used to define a method. Bold words are predefined terminal symbols that are part of the language, italic words are used-defined identifiers and ::=, |, + and * are part of the BNF formalism.

as functions over sets of entities. The second way to define a feature value is describing a new entity-description. For instance, the `owner` feature value of `car` can be defined as a refinement of the entity `person`. (see Figure 1). The third way to define feature values is by means of method descriptions.

2.2.1 Methods

Methods are also mapped to the language as entity descriptions. 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.

Examples of NOOS built-in methods are arithmetic operations, set operations, logic operations, operations for comparing entities and other basic constructs such as conditional or sequencing. For each built-in method a set of built-in features is defined. 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`. In Figure 1 the built-in `identity?` method is used to describe the `empty-level?` feature value 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).

An important subset of built-in methods are *query-methods*. These methods establish a correspondence between two feature values of some entities. For instance, in our car diagnosis domain the `gas-gauge-reading` feature value of `Car` is defined by a *reference* to the `gas-level-in-tank`

```
(define (conditional Causal-Explanation)
  (cause )
  (effect )
  (condition (>> cause))
  (result (>> effect)))

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

(define (decomposition-method Generate&Test)
  ((generate generate-method-1))
  ((test test-method-2)))
```

Figure 3: Definition of entity methods `causal-explanation`, `C1` as a refinement of `causal-explanation`, and `Generate&Test` method as a refinement of `decomposition-method`.

feature value. The meaning of this reference is that `gas-gauge-reading` feature value is constrained to be the same as `gas-level-in-tank` feature value (see Figure 1).

NOOS language provides a special syntax for the description of references by means of query-methods. There are two ways to define references: absolute references and relative references. An absolute reference is `(>> feature of entity)` where `feature` and `entity` stand for 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. For instance, in Figure 1 the roots are `car`, `Ibiza-car` and `Peters-car`.

A reference to another feature value of some entity can be established indirectly by means of intermediate feature references. This composition of feature references is called *path*. For instance, the `price` of a given `car` can be established as the `price` of the `model` of this `car`, writing `(>> price model)` in the `price` feature of `car` (see Figure 1).

As we have shown methods are defined in NOOS language by refinement. In order to illustrate the definition of methods in the NOOS language we will introduce an example: a `causal-explanation` method is defined as a refinement of the built-in `conditional` method where `condition` and `result` features (the built-in features defined for `conditional`) are defined as a reference to `effect` and `cause` feature values respectively. Next, concrete causal explanations can be defined through refinements of `causal-explanation` – e.g. causal explanation `C1` in Figure 3. Causal explanation `C1` justifies the conclusion of having a `low-battery-malfunction` when the `battery-voltage` feature value of a given `car` is equal to `low-voltage`.

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. In Figure 3 `generate&test` method is defined using references to `generate-method-1` and `test-method-2` methods defined elsewhere. Section 4 shows a more detailed generate and test method for diagnosis.

2.2.2 Metalevels

Metalevels in the NOOS model are also mapped to *entity descriptions* in the NOOS language.

A metalevel description is just an entity description plus a metalevel relation with a (base-level) entity (called *referent* entity of the metalevel entity). The features defined in the body of a metalevel entity description have a corresponding feature in the base-level with the same name (and vice versa). Since metalevels are defined in the same way that concepts and methods, feature values of metalevel features can be defined by reference to other entities or metalevels, defining a path or defining a (metalevel) method.

The definition of feature values in metalevels by means of references allows to define directly a set of alternative methods for a given feature. Moreover, a partial order among this set of methods can be defined. This partial order will be interpreted as a preference ordering over the alternative methods. This process will be explained in section 2.3. Metalevel feature values can be defined with a path, allowing a metalevel entity to refer to some methods described in another entity. The last way to define a metalevel feature value is by means of a (metalevel) method. A metalevel method computes a set of ordered methods for that metalevel feature. This metalevel method can take into account the information given in the current problem. We have shown elsewhere that case-based reasoning methods [2] and inheritance [14] can be defined as metalevel methods in NOOS. Another example of metalevel methods given in [3] shows the use of a generate and test strategy for selecting (from a set of possible hypotheses) causal methods used in previous problems, according to the knowledge available in the current problem.

The definition of metalevel entities is always a refinement of the `meta` entity. For instance, the following example defines a metalevel entity that has as referent the `Car` entity and has two feature descriptions.

```
(define (Metalevel (meta of Car))
  ; these are methods for empty-level?  features of 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)
                 (current-complaint (>> complaint of (referent)))))) ; see footnote[1]
```

In the metalevel feature `empty-level?` a set of two alternative methods are given. In the metalevel `diagnosis` feature a metalevel method that infers a set of alternatives methods is defined. Specifically, `select-car-explanations` is a metalevel method that according to the current complaint information of a car will provide a set of partially ordered explanation methods for the `diagnosis` feature.

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 (cases or previously solved problems). An example of CBR method is defining `select-car-explanations` method as a retrieval method. This method can examine the previous car problems and retrieve those that have in common with current problem at least the same complaint. Then, it selects the explanation methods that were successfully used in the `diagnosis` task of those cases as the best possible explanations for the current diagnosis problem [2].

Since metalevels and methods are also defined by means of entity descriptions, this uniformity has two main consequences for the expressive power of NOOS. First, since a metalevel is also an entity, a metalevel can also have its own metalevel. Thus, virtually infinite towers of metalevels are allowed. Second, since methods are entity descriptions with a set of features interpreted as subtasks, the metalevel feature description of a subtask allows to define multiple methods to achieve that subtask. In the example below, a metalevel for the `generate-and-test` method is defined. In the `generate` subtask two different methods for generating hypotheses are given. The `test` subtask also has two methods for testing the generated hypothesis.

[1] The expression `(referent)` is reference to the base-level entity that is the referent of the metalevel where it occurs. In this case, `referent` of `meta` of `Car` refers to `Car`.

```
(define (Metalevel (meta of Generate&Test))
  (generate generate-hypothesis-method-1
            generate-hypothesis-method-2)
  (test test-method-1 test-method-2))
```

2.3 Inference in Noos

Inference in NOOS is on demand. Thus, inference starts when a user asks to solve a specific task by means of *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 chaining ends when a task has a constant value. A task is achieved when its corresponding method is successful, and a method is successful when all its subtasks are achieved.

A query expression is interpreted in two steps: first it is reified as a method and then this method is evaluated. For instance, the query expression `(>> diagnosis of Peters-car)` is reified into the query-method `<infer-value_33>` and then `<infer-value_33>` is evaluated, eventually producing the solution for `Peters-car` diagnosis.

When no method is specified for a given task, an *impasse* occurs and the control of the inference is passed to the corresponding task at the metalevel. The task at the metalevel has to infer a partially ordered set of alternative methods for the current task by means of a metalevel method. This 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 an achieved overall task, the combination of all successful methods in its tree of task/method decomposition will be maximal with respect to the preference orders inferred by the metalevels tasks involved. Formally,

Definition 1 (Maximal solution) *Given the set of achieved subtasks $t_1, t_2 \cdots t_n$, that form the task decomposition of a main task T, given the set of partial orders $\prec_1, \cdots \prec_n$ over the alternative methods for these subtasks, and given the set of methods $m_1, m_2 \cdots m_n$ engaged respectively to these subtasks, a solution of T is* maximal *if there is no other combination of methods $m'_1 \prec_1 m_1$, $m'_2 \prec_2 m_2, \cdots m'_n \prec_n m_n$ (where at least one $m'_i \neq m_i$) that achieves a solution for T.*

The definition just given is indeterministic when the maximal is not unique, and corresponds to the formalization developed in section 3.5. The interpreter of NOOS uses backtracking to search for a method combination such that is maximal with respect to the preferences involved.

2.3.1 Metalevel Inference

He have introduced the notion of query-methods in section 2.2.1 . In fact, NOOS provides four kinds of query-methods: *Infer-value*, *Known-value*, *Exists-value* and *All-values*. They correspond to the definition *op* in figure 2, namely `>>`, `!>>`, `?>>` and `*>>` respectively. The semantics of *Infer-value* method corresponds to the examples showed in the previous section: feature value equality. The rest three query-methods provides a set of basic metalevel inference capabilities about feature values of entities. The *Known-value* method establishes a feature value equality only when the feature value is already known (without engaging inference). *Exists-value* is a query-method that determines whether there is any inferrable value for a given feature of an entity. The result of *Exists-value* is `true` if there is at least one method successful in achieving that task, and `false` otherwise. *All-values* is a query-method that determines the set of all inferrable values for a given feature of an entity – the set of results of all methods successful in achieving that task.

In NOOS the process of method evaluation can be reified also in the language. Method evaluation is reified by means of the *noos-eval* method. For instance, the query expression `(>> diagnosis of Peters-car)` has a meaning that is equivalent to

```
(noos-eval (reify (>> diagnosis of Peters-car)))
```

In turn, this query expression is reified into `noos-eval` method with a feature `method` whose value is the query-method corresponding to the reification of the original query expression

```
(define (noos-eval)
  (method (reify (>> diagnosis of Peters-car))))
```

The `reify` operator constructs a query-method from a query expression. So the former expression is equivalent to

```
(define (noos-eval)
  (method (define (Infer-value)
            (feature 'diagnosis)
            (domain   Peters-car))))
```

In order to provide a set of metalevel inference capabilities about method evaluation, four evaluation-methods are defined corresponding to the four existing query-methods: *Noos-eval*, *Known-eval*, *Exists-eval* and *All-eval*. *Noos-eval* performs the *method evaluation process* previously explained. The rest of three evaluation-methods are built on top of this basic method evaluation process. *Known-eval* succeeds only when the result of the method evaluation is already known, and returns that value. *Exists-eval* determines whether it is possible to evaluate successfully the method; the result will be `true` if it is possible and `false` otherwise. Finally, *All-eval* returns the set of results of all the successful evaluations of the method.

3 Noos Formalization

We will describe formally the inference in our system using Descriptive Dynamic Logic [16]. Descriptive Dynamic Logic (DDL) is a propositional dynamic logic (PDL) [9] to describe architectures to build reflective knowledge-based systems with complex reasoning patterns developed also at our Institute.

3.1 Descriptive 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 bridge rules, to specify the information flow among the different units of the RKB. For a full description of DDL see [16]. Here we only present some basic definitions of DDL used later:

Definition 2 *A Multi-Language Logical Architecture is a 4-tuple* $MLA = (L, \Delta, S, T)$*, where:*

1. $L = \{L_j\}_{j \in J}$ is a set of logical languages.
2. $\Delta = \{\Delta_{j_1,j_2}\}_{j_1,j_2 \in J}$ is a set of (instances of) inference rules between pairs of languages, i.e $\Delta_{j_1,j_2} \subseteq 2^{L_{j_1}} \times L_{j_2}$. In particular, when $j_1 = j_2$, Δ_{j_1,j_2} denotes a set of inference rules of the corresponding language; otherwise it denotes a set of bridge rules between two different languages.
3. S is a finite set of symbols for unit identifiers.
4. T is the set of possible topologies. Each topology is determined by a set of directed links between symbols from S, i.e T is a subset of $2^{S \times S}$.

Definition 3 *A Reflective Knowledge-Based System RKB for a given MLA is a 7-tuple* $RKB = (MLA, U, M_L, M_\Delta, B, M_\Sigma, M_\Omega)$ *where: U is a set of unit identifiers. M_L assigns a language to each unit identifier. M_Δ assigns a set of inference rules to each unit identifier. B is a mapping that assigns a set of directed bridge rules to pairs of different units in accordance with the allowed topologies in RLA. M_Σ assigns a concrete signature for the language of each unit identifier. M_Ω assigns a set of formulas (initial local theory) to each unit identifier.*

Definition 4 *The set Φ_0 of atomic formulas of DDL will be defined as the set of "quoted" formulas from the languages L in MLA, indexed by the unit identifiers in U.*

$$\Phi_0 = \{u : \lceil\varphi\rceil | u \in U, \varphi \in M_L(u)\}$$

Definition 5 *The set Π_0 of atomic programs of DDL is defined as the union of intra-unit inference rules Π_0^{Intra} and the inter-unit rules Π_0^{Inter}*

$$\Pi_0 = \left(\bigcup_{k \in K} \Pi_{0_{kk}}^{Intra}\right) \cup \left(\bigcup_{k \neq j \in K} \Pi_{0_{kl}}^{Inter}\right)$$

Definition 6 *The DDL semantics, following the PDL semantics, is defined relative to a structure M of the form $M = (W, \tau, \rho)$, where W is a set of states, τ a mapping $\tau : \Phi \rightarrow 2^W$ assigning to each formula φ the set of states in which φ is true, and ρ a mapping $\rho : \Pi \rightarrow 2^{W \times W}$ which assigns to each program a set of pairs (s,t) representing transitions between states.*

3.2 Noos Unit Languages

Formally, every NOOS entity is represented as a DDL unit. There are three kinds of unit languages: *concept languages*, *method languages* and *metalevel languages*. Concept languages pertain to units representing concepts (called concept units). Method languages pertain to units representing methods (called method units) and are an extension of concept languages. Metalevel languages pertain to units representing metalevels (called metalevel units) and are also an extension of concept languages. The signature Σ_c of the language of a concept unit c is defined as

$$\Sigma_c = \Sigma_c^e \cup \Sigma_c^f \cup \{\emptyset, true, false\}$$

where Σ_c^e is a set of symbols of entities, Σ_c^f is a set of symbols of features, and $\{\emptyset, true, false\}$ are empty set, true and false symbols respectively.

The set of terms Γ_c of the language of a concept unit c is formed according to the following rules:

$\Sigma_c^e \subset \Gamma_c$
$\emptyset, true, false \in \Gamma_c$
$c_1, \cdots, c_i \in \Sigma_c^e \cup \{true, false\} : \{c_1 \cdots c_i\} \in \Gamma_c$
$\{c_1 \cdots c_i\} \in \Gamma_c, \prec$ is a pre-order defined on $\{c_1 \cdots c_i\} : \{c_1 \cdots c_i | \prec\} \in \Gamma_c$

The set of formulas Φ_c of a given concept unit c contains formulas describing the feature values pertaining to each feature and formulas describing the method pertaining to each feature.

$$f \in \Sigma_c^f, t \in \Gamma_c, m \in \Sigma_c^e : \quad \begin{array}{l} f \doteq t \in \Phi_c \\ f \doteq \#m \in \Phi_c \end{array}$$

The language of a metalevel unit μ is a concept unit language extended with formulas describing the set of formulas about features contained in its referent unit. Thus, a metalevel unit μ contains formulas describing for each feature in the referent unit the method and value (referent) pertaining to the feature.

$$f \in \Sigma_\mu^f, m \in \Sigma_\mu^e : method(f) \doteq m$$

$$f \in \Sigma_\mu^f, t \in \Gamma_\mu : referent(f) \doteq t$$

The set of formulas Φ_m of a given method unit m contains also formulas with feature values and feature methods like concept units. Moreover, Φ_m contains a set of formulas describing the result of the method evaluation

$$t \in \Gamma_m : result(m) \doteq t \in \Phi_m$$

As we have shown, query-methods are a special kind of methods that provide metalevel capabilities of reasoning about feature values of entities. Query-methods are also represented as DDL units. Their languages are method languages enriched with formulas containing the feature values of features in other entities. Thus, the set of formulas Φ_m of a given query-method unit m is enriched by

$$f \in \Sigma_m^f, c_i \in \Sigma_m^e, t \in \Gamma_m : c_i.f \doteq t \in \Phi_m$$

Another special kind of methods are eval-methods. Their languages are method languages enriched with formulas containing the evaluation results of other methods.

$$m_i \in \Sigma_m^e, t \in \Gamma_m : result(m_i) \doteq t$$

3.3 Inference Rules

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

Definition 7 *Every entity is considered equivalent to the singleton set that contains this entity.*

Definition 8 *A set of entities is considered equivalent to a partially ordered set with the same entities and no ordering among them.*

3.3.1 Intra-Unit Inference Rules

Only methods and metalevel units have intra-unit inference rules. Inference rules in metalevel units select one method from a set of alternative methods; latter reflection rules will reflect this selected method to its referent unit. The inference rule for method selection is:

$$\delta_f^{select} = \frac{\begin{array}{c} f \doteq \{S|\prec\} \\ m \in S \end{array}}{method(f) \doteq m}$$

Inference rules in methods code the built-in definition of the evaluation of the method. There is one inference rule for each type of built-in method provided by NOOS. For instance, the inference rule for an add-method m is defined as

$$\delta_m^{add} = \frac{\begin{array}{c} item1 \doteq c_1 \\ item2 \doteq c_2 \\ \text{“}c' = c_1 + c_2\text{”} \end{array}}{result(m) \doteq c'}$$

where a new formula $result(m) \doteq c'$ is added with the result of the sum of the feature values given in $item1$ and $item2$.

A more interesting inference rule is the inference rule for query-methods that allows to reason about feature values of other units. The inference rule δ_m^{iv} for a query-method m is defined as

$$\delta_m^{iv} = \frac{\begin{array}{c} feature \doteq f \\ domain \doteq \{c_1 \cdots c_n|\prec\} \\ c_1.f \doteq s_1 \\ \cdots \\ c_n.f \doteq s_n \end{array}}{result(m) \doteq \{\cup s_i|\prec'\}}$$

where $\prec'$ is a partial order over the new resulting set $\{\cup s_i\}$ that translate the existing $\prec$ order in this way:

$$\prec_f = \overline{\{(x,y) | \exists (c_i, c_j) \in \prec, x \in s_i, y \in s_j\}}$$

and $\overline{R}$ represents the closure of relation R.

Another important inference rule is the rule for eval-methods that allows to reason about method evaluations. The inference rule δ_m^{ne} for an Eval-method m is defined as

$$\delta_m^{ne} = \frac{\begin{array}{c} methods \doteq \{m_1 \cdots m_n | \prec\} \\ result(m_1) \doteq s_1 \\ \cdots \\ result(m_n) \doteq s_n \end{array}}{result(m) \doteq \{\cup s_i | \prec'\}}$$

where $\prec'$ is a partial order defined as in the previous query-method rule.

3.3.2 Inter-Unit Inference Rules

There are four kinds of inter-unit inference rules: *Reification rules*, *Reflection rules*, *Reduction rules*, and *Translation 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. *Reduction rules* add to an unit the result of the evaluation of one of its methods. Finally, *Translation rules* specify how formulas may be transported from a unit to another one.

Reification rules $\delta_{c\mu}^{up}$ add to the metalevel unit μ the set formulas about the feature values known in the unit c

$$\delta_{c\mu}^{up} = \frac{f \doteq t}{referent(f) \doteq t}$$

Reflection rules $\delta_{\mu cf}^{down}$ add to the base-level unit c a formula about the feature method selected by the metalevel unit μ

$$\delta_{\mu cf}^{down} = \frac{method(f) \doteq m}{f \doteq \#m}$$

Reduction rules δ_{mcf}^{red} add to an unit c the formula for feature f with the result of the evaluation of one method unit m.

$$\delta_{mcf}^{red} = \frac{result(m) \doteq t}{f \doteq t}$$

Translation rules δ_{cmf}^{tquery} add from an unit c to a query-method unit m the formula for the feature f (section 3.5 illustrates their use in query-method evaluation).

$$\delta_{cmf}^{tquery} = \frac{f \doteq t}{c.f \doteq t}$$

Translation rules $\delta_{m_1 m_2}^{teval}$ add from a method unit m_1 to a method unit m_2 a formula with the evaluation result of m_1

$$\delta_{m_1 m_2}^{teval} = \frac{result(m_1) \doteq t}{result(m_1) \doteq t}$$

3.4 Topology

The set of possible topologies in NOOS is formed by three kinds of relations among units: *reference relations*, *feature method relations* and *metalevel relations*. The set of reference relations of an entity c with other entities c' is determined by the set of formulas $f \doteq c'$ contained in c. The set of feature method relations of an entity c with other method entities m is determined by the set of formulas $f \doteq \#m$ contained in c. Metalevel relations are by explicit `meta` relations in NOOS programs. Metalevel relations are exclusive relations: one unit can be the metalevel of only another unit (called *referent*), one unit can only have one metalevel unit, and cycles are forbidden. Thus, a metalevel entity can be the referent of another (meta)metalevel entity.

3.5 Programs

The set Π_c^{Intra} of programs for a unit c is defined as the set of query programs $\pi_{c.f}$ of the features f defined in Σ_c :

$$\Pi_c^{Intra} = \bigcup_{f \in \Sigma_c} \pi_{c.f}$$

where $\pi_{c.f}$ is defined as follows

$$\pi_{c.f} = (\pi_{c.f}^{\mu} \cup true?); \bigcup_{m \in \Sigma_c} ((f \doteq \#m)?; \pi_m; \delta_{mcf}^{red})$$

The query program $\pi_{c.f}$ for a feature f of unit c is defined as the sequential concatenation[2] of three programs: i) the metalevel inference program $\pi_{c.f}^{\mu}$, ii) the evaluation program π_m of the method m, and iii) the inference rule δ_{mcf}^{red} that adds a new formula with the result of the method evaluation. The indeterministic union $\pi_{c.f}^{\mu} \cup true?$ expresses the possibility to skip the metalevel inference step when there is a method defined in the unit c.

The metalevel inference program $\pi_{c.f}^{\mu}$ starts with a reification inference rule δ_c^{up}, then engages the query program $\pi_{\mu.f}$ for a feature f at the metalevel unit μ, selects one of the methods obtained in the previous step (δ_f^{select}), and reflects down this selected method to the referent unit ($\delta_{\mu cf}^{down}$).

$$\pi_{c.f}^{\mu} = \delta_c^{up}; \pi_{\mu.f}; \delta_f^{select}; \delta_{\mu cf}^{down}$$

Evaluations of methods in NOOS are also formalized as DDL programs:

$$\pi_m = \pi_{m.f_1}; \cdots; \pi_{m.f_n}; \delta_m$$

The evaluation program π_m of a method m is composed by the sequence of query programs to their subtasks $(f_1, \cdots, f_n)$ followed by the program of the intra-unit inference rule of m (that combines the partial results of subtasks into a final result). For instance, the evaluation program π_m^{add} of an add method m is the sequence of query programs to compute the operands and the intra-unit rule δ_m^{add} that combines them.

$$\pi_m^{add} = \pi_{m.item1}; \pi_{m.item2}; \delta_m^{add}$$

Query-methods have also their own evaluation programs. The evaluation program of a query-method is the reification of inference in NOOS. A query-method m involves the subtask *feature* (that computes the feature name f of the query), the subtask *domain* (that computes the entity or set of entities s to which the query is addressed), and finally that query f is performed to all entities s. We use s as a shorthand of $\{c_1 \cdots c_n | \prec\}$. The first query-method is *Infer-value*:

$$\pi_m^{iv} = \pi_{m.feature}; \pi_{m.domain}; \bigcup_{s,f \in L_m} (R_m^{iv}, \delta_m^{iv})$$

[2] The concatenation of programs is expressed by ';' in PDL syntax. '$\phi?$' denotes the program that evaluates whether a given formula ϕ is true. $\langle\phi\rangle$ and $[\phi]$ are interpreted as the usual modal operators.

where

$$R_m^{iv} = ((feature \doteq f) \wedge (domain \doteq \{c_1 \cdots c_n | \prec\}))?; \pi_{c_1.f}; \delta^{query}_{c_1 m f}; \cdots; \pi_{c_n.f}; \delta^{query}_{c_n m f}$$

The evaluation program π_m^{iv} of an *Infer-value* method m first engages the computation of the name f of a feature ($\pi_{m.feature}$), next computes the entity or entities s to which the query is addressed ($\pi_{m.domain}$), and finally the query f is performed to all entities s (R_m^{iv}) and the results are combined by δ_m^{iv}.

The evaluation program π_m^{kv} of a *Known-value* method m is analogous to the previous evaluation program except that no inference is engaged:

$$\pi_m^{kv} = \pi_{m.feature}; \pi_{m.domain}; \bigcup_{s,f \in L_m} (R_m^{kv}; \delta_m^{iv})$$

where

$$R_m^{kv} = ((feature \doteq f) \wedge (domain \doteq \{c_1 \cdots c_n | \prec\}))?; \delta^{query}_{mc_1 f}; \cdots; \delta^{query}_{mc_n f}$$

Notice that the R_m^{kv} program is composed only by translation rules. Thus, the evaluation of *Known-value* methods will be completed only when all feature values have been already inferred.

The evaluation program π_m^{ev} of an *Exists-value* method m checks whether any solution to a query exists, and returns a boolean accordingly:

$$\pi_m^{ev} = \underline{\text{if}} \langle \pi_m^{iv} \rangle ? \ \underline{\text{then}} \ \delta_m^{true} \ \underline{\text{else}} \ \delta_m^{false}$$

where δ_m^{true} and δ_m^{false} are just intra-unit inference rules assigning *true* and *false* respectively.

Finally, evaluation program π_m^{av} of *All-values* method determines the set of all inferrable values:

$$\pi_m^{av} = \pi_{m.feature}; \pi_{m.domain}; \bigcup_{s,f \in L_m} (R_m^{av}; \delta_m^{av})$$

and

$$R_m^{av} = ((feature \doteq f) \wedge (domain \doteq \{c_1 \cdots c_n | \prec\}))?; (\pi_{c_1.f}; \delta^{iv}_{mc_1 f})^c; \cdots; (\pi_{c_n.f}; \delta^{iv}_{mc_n f})^c$$

where α^c represents the closure of program α – that is, this program will lead to a state in which no different state is reachable by another application of program α. Specifically, a closure $(\pi_{c_i.f}; \delta^{iv}_{mc_i f})^c$ gathers all possible values of query program $\pi_{c_i.f}$.

Query methods deal with the methods of a specific task and determine which of the 4 kinds of inference is engaged by that task. In order to deal with a specific method entity, NOOS uses 4 kinds of eval-methods.

The evaluation program of an eval-method m involves the subtask *methods* that engages in the computation of the methods to be evaluated. Next, the evaluation programs of these methods are performed. The first eval-method is *Noos-eval*:

$$\pi_m^{ne} = \pi_{m.methods}; \bigcup_{s \in L_m} (R_m^{ne}; \delta_m^{ne})$$

where

$$R_m^{ne} = (methods \doteq \{m_1 \cdots m_n | \prec\})?; \pi_{m_1}; \delta^{ne}_{m_1 m}; \cdots; \pi_{m_n}; \delta^{ne}_{m_n m}$$

The evaluation program π_m^{ne} of an eval-method m first engages the computation of the methods to be evaluated ($\pi_{m.methods}$), next performs the evaluation programs of these methods (R_m^{ne}), and finally the results are combined (δ_m^{ne}).

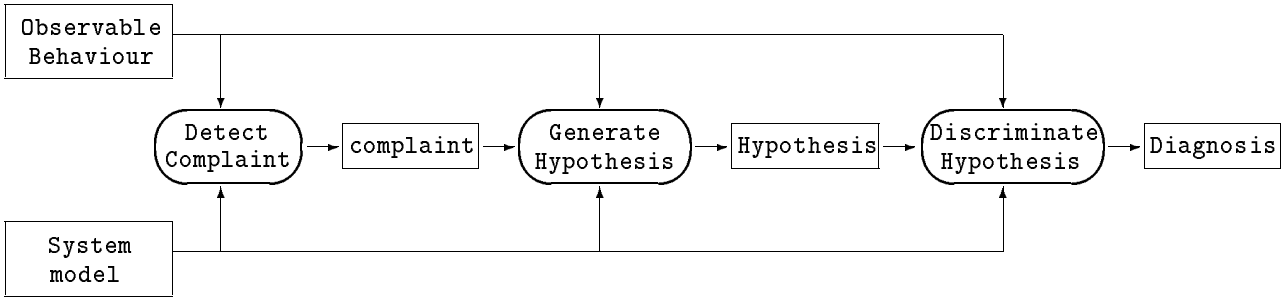

Figure 4: General scheme for diagnosis task.

The evaluation program of a *Known-eval* method m is analogous to the previous evaluation program except that no inference is engaged:

$$\pi_m^{ke} = \pi_{m.methods}; \bigcup_{s \in L_m} (R_m^{ke}; \delta_m^{ne})$$

where

$$R_m^{ke} = (methods \doteq \{m_1 \cdots m_n | \prec\})?; \delta_{m_1 m}^{ne}; \cdots; \delta_{m_n m}^{ne}$$

The evaluation program of an *Exists-eval* method m checks whether any solution to evaluation exists, and returns a boolean accordingly:

$$\pi_m^{ee} = \underline{\text{if}} \langle \pi_m^{ne} \rangle? \ \underline{\text{then}} \ \delta_m^{true} \ \underline{\text{else}} \ \delta_m^{false}$$

Finally, the evaluation program of a *All-eval* method m infers all the possible values:

$$\pi_m^{ae} = \pi_{m.methods}; \bigcup_{s \in L_m} (R_m^{ae}; \delta_m^{ae})$$

where

$$R_m^{ae} = (methods \doteq \{m_1 \cdots m_n | \prec\})?; (\pi_{m_1}; \delta_{m_1 m}^{ne})^c; \cdots; (\pi_{m_n}; \delta_{m_n m}^{ne})^c$$

4 An example for diagnosis tasks

The purpose of this section is to show the NOOS computational support using reflection to knowledge modelling. We will use as example the knowledge modelling analysis of diagnosis tasks performed by R. Benjamins [7]. The original analysis was intended for the knowledge modelling framework KADS [19] developed as a methodology for knowledge acquisition.

Diagnosis task can be decomposed in three general subtasks: `Detect-Complaint`, `Generate-Hypothesis` and `Discriminate-Hypothesis` (The general scheme for diagnosis task is showed in Figure 4).

The first `Detect-Complaint` task may identify the complaint using one of several alternative methods (see figure 5). The easiest method is directly asking the user, but classification methods or comparison methods could also be used.

The next two subtasks of diagnosis are `Generate-Hypothesis` and `Discriminate-Hypothesis`. As shown in figure 5, there are several alternative methods for these tasks. Essentially, these methods are different because they use knowledge of different kinds of models we may have of a system, like behavior models, associations models and causal dependency models. For instance, in order to achieve the `Generate-Hypothesis` task two different methods – namely `model-based-hypothesis-generation` and `empirical-hypothesisgeneration`

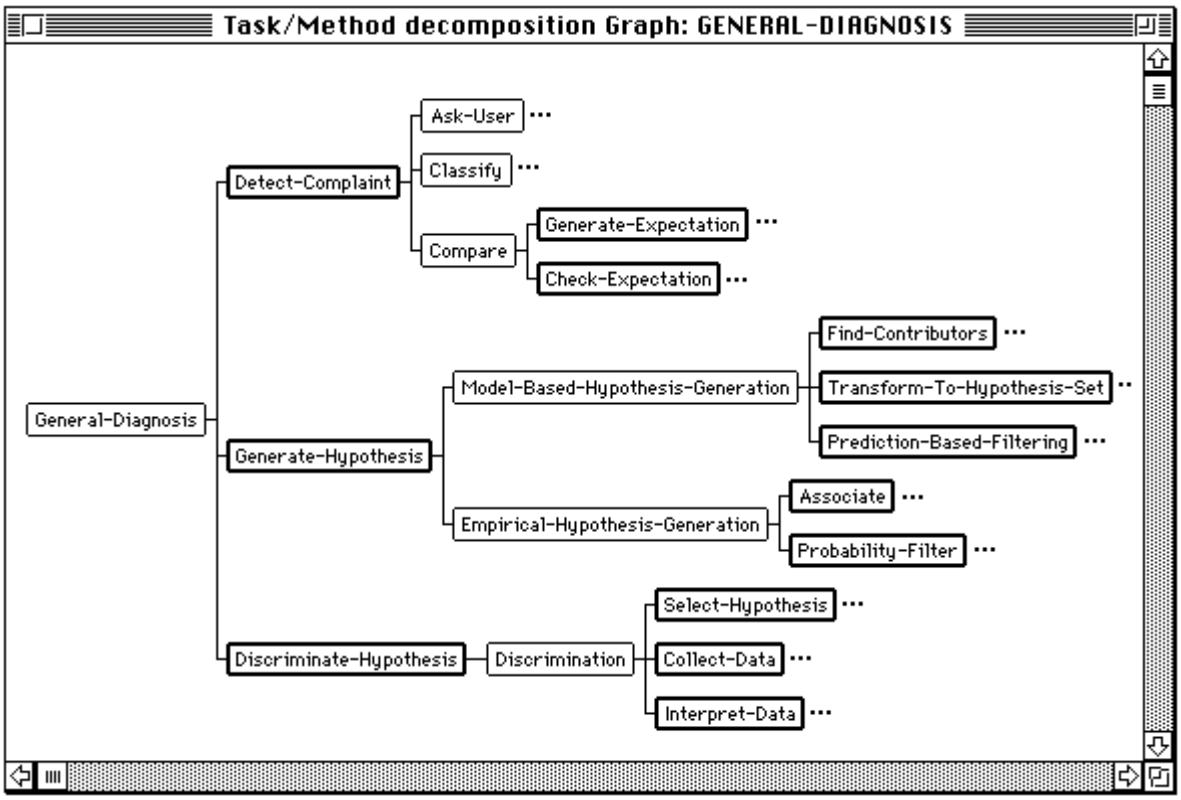

Figure 5: A browser of partial task/method decomposition for general diagnosis method. Tasks are painted with thin boxes and methods are painted with thick boxes

– can be used (see Fig 5). The first of them requires a device-model of the domain while `empirical-hypothesis-generation` method requires a domain with empirical associations. `Model-based-hypothesis-generation` method decomposes successively the generate-hypothesis task in three new subtasks: `find-contributors`, `transform-to-hypothesis-set` and `prediction-based-filtering`. Notice that multiple alternative methods to perform a task can be defined, each method is a NOOS entity that defines its specific subtasks and each of those subtasks may also have successively multiple alternative methods.

Another kind of knowledge that can be useful is that which Benjamins [7] called the suitability criteria: "A suitability criterion represents the applicability of a method. It is a requirement reflecting features of a method, that are relevant for determining the method appropriateness". This is a kind of strategic knowledge that can be represented at the metalevel in the NOOS language. A metalevel method can be used in order to infer which of the alternative methods for a given task is more appropriate to a specific problem description, based on knowledge about the method suitabilities and the particular information available at a given problem.

In the present example we will design a metalevel method for diagnosis task that chooses among the different alternative methods available for diagnosis: `choose-diagnosis-method`. First of all, metalevel method `choose-diagnosis-method` has to have some knowledge *about* the diagnosis methods. In order to represent this, `choose-diagnosis-method` will work upon `model-of-method` entities. A `model-of-method` holds the relevant information to characterize a method (i.e. is a *model of* the method). Specifically, a `model-of-method` holds a method in the `method` feature and in other features holds information about the `principle` on which this method is based (e.g. model-based or empirical), and the `requirements` that the method has upon the knowledge it needs to have available in order to be able to achieve a solutions. Moreover, since the decisions will depend upon the specific problem to be solved, a `problem` feature is present in the

```
(define (model-of-method MBR-Method-Model)
  (problem )
  (method (define (model-based-hypothesis-generation)
            (problem (>> problem))))
  (principle model-based-reasoning)
  (requirements (or (?>> trace-model problem)
                    (?>> causal-model problem)
                    (?>> prediction-model problem))))

(define (model-of-method Empirical-Method-Model)
  (problem )
  (method (define (empirical-hypothesis-generation)
            (problem (>> problem))))
  (principle empirical-association)
  (requirements (?>> associate-model problem)))
```

Figure 6: Models of model-based reasoning (MBR) and empirical methods.

`model-of-method` entity.

Figure 6 shows the models of methods `model-based-hypothesis-generation` and `empirical-hypothesis-generation`. These two methods are the key to perform a diagnosis based either on device-models or on empirical association. Both methods generate hypotheses of malfunctions. Their suitability depends on being able to determine whether or not the models they require are present in a specific problem. To check whether these models are already present or can be inferred, the *Exists-value* query-method '`?>>`' is used in the `requirements` feature of the models.

Metalevel method `choose-diagnosis-method` can be now defined manipulating these models of methods, as shown in Figure 7. First, a preference of model-based reasoning upon the rest of more empirical methods is set, since model-based diagnosis tends to be more reliable than that based on empirical methods.

Preference method `Subsumption-Preference` sets a preference for those values in `sources` that satisfy a `pattern` over those that do not satisfy it. In this case, the pattern is satisfied by `model-of-method`s whose `principle` is `model-based-reasoning`. As result, `method-models` feature of holds a poset with `mbr-method-model` being preferred to `empirical-method-model`. In `preferred-model` task the `reify` operator lifts this poset to the metalevel of this same task. The operational semantics of the reflection cycle (section 3.5) will reflect down from the metalevel one of the most preferred models, in this case there is one most preferred, namely `mbr-method-model`. The `condition` task of `choose-diagnosis-method` checks whether the `requirements` of `preferred-model` hold. In case these `requirements` hold, `mbr-method-model` will be the preferred model. In case these `requirements` do not hold, the system backtracks the last decision to reflect down the most preferred item and reflects down the next preferred – which would be `empirical-method-model`. Finally, the `result` task selects the `method` of the model of method that is the `preferred-model`.

Now `choose-diagnosis-method` can be defined as the metalevel method for the `diagnosis` task, as shown in figure 8. First, a metalevel `diagnosis-meta` is defined in which `choose-diagnosis-method` is the metalevel method for the `diagnosis` task. Next, a metalevel of `diagnosis-problem` is defined as refinement of `diagnosis-meta`. Finally, any specific problem can be defined as a refinement of `diagnosis-problem`.

```
(define (conditional-method Choose-Diagnosis-Method)
  (problem )
  ((method-models (define (Subsumption-Preference)
                    (pattern (define (model-of-method)
                               (principle model-based-reasoning)))
                    (sources (define (mbr-method-model)
                               (problem (>> problem)))
                             (define (empirical-method-model)
                               (problem (>> problem)))))))
  ((preferred-model (reify of (>> method-models))))
  (condition (>> requirements preferred-model))
  (result (>> method preferred-model)))
```

Figure 7: The definition of a method that selects the diagnosis method most suitable to a given problem.

```
(define (meta Diagnosis-Meta)
  ((diagnosis (define (choose-diagnosis-method)
                (problem (referent))))))

(define Diagnosis-Problem)

(define (diagnosis-meta (Meta of Diagnosis-Problem))

(define (diagnosis-problem Problem-13)
  (complaint does-not-start)
  ...)
```

Figure 8: `Choose-Diagnosis-Method` is installed as a metalevel method for the diagnosis task.

5 Related work and Conclusions

Metalevel architectures in AI systems have been used for a wide range of purposes: strategic reasoning [8] [11], non-monotonic reasoning [18], and modelling expert systems [1]. Precedents on using reflection for case-based reasoning are [10] and [15]. Our work on architectures is related to cognitive architectures that integrate learning with problem solving like SOAR [13], THEO [12]. SOAR learning is based on a single method called *chunking* while our purpose in NOOS has been to integrate several learning methods within a problem solving framework. THEO integrates several learning methods but provides more restricted metalevel capabilities than NOOS – the metalevel methods allowed are predefined and ranked by a total order. Related work on knowledge-level modelling of AI systems includes the COMMET (or components of expertise) framework [17], and the KADS methodology [1]. 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 model of entities, tasks, and methods.

The design of NOOS was guided by the goal of following knowledge modelling analysis in KBS construction and the integration of learning methods in the same framework [3]. We have performed a knowledge modelling analysis of case-based reasoning and learning [5] and we have used NOOS to implement CHROMA [6], 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 class of marine sponge species (the family of *Geodiidae*). SPIN currently integrates a bottom-up induction method, a top-down induction method, and an CBR method based on an entropy measure.

We have shown in this paper the role of preferences in section 2.3 and in the DDL formalization. However, complex reasoning about preferences has not been explained. In particular, NOOS supports operations for combining multiple preferences, constructors of preferences and higher order preferences [4].

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 [8] and BOLERO [11] 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 architecture for problem solving and learning.

Acknowledgements

The research reported on this paper has been developed at the IIIA inside the ANALOG Project funded by Spanish CICYT grant 122/93, the European TMR project *VIM* (PL93-0186) and with a CSIC fellowship for the first author.

References

[1] 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.

[2] Josep Lluís Arcos and Enric Plaza. A reflective architecture for integrated memory-based learning and reasoning. In S. Wess, K.D. Althoff, and M. Richter, editors, *Topics in Case-Based Reasoning*, number 837 in Lecture Notes in Artificial Intelligence, pages 289–300. Springer-Verlag, 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. Reasoning with preferences in a reflective framework. 1996. (submited).

[5] Eva Armengol and Enric Plaza. A knowledge level model of case-based learning. In S. Wess, K.D. Althoff, and M. Richter, editors, *Topics in Case-Based Reasoning*, number 837 in Lecture Notes in Artificial Intelligence, pages 53–64. Springer-Verlag, 1993.

[6] 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.

[7] Richard Benjamins. On a role of problem solving methods in knowledge acquisition - experiments with diagnostic strategies. In Luc Steels, Guss Schreiber, and Walter Van de Velde, editors, *A Future for Knowledge Acquisition*, number 867 in Lecture Notes in Artificial Intelligence. Springer-Verlag, 1994.

[8] Lluís Godo, Ramon López de Mántaras, Carles Sierra, and Albert Verdaguer. Milord: The architecture and the management of linguistically expressed uncertainty. *Int. J. Intelligent Systems*, 4:471–501, 1989.

[9] D. Harel. Dynamic logic. In D. M. Gabbay and F. Guenthner, editors, *Handbook of Phylosophical Logic*, volume 2, pages 497–604. Kluwer, 1984.

[10] Beatriz López and Enric Plaza. Case-based learning on strategic knowledge. In Y. Kodratoff, editor, *Machine Learning-European Working Session on Learning-91*, volume 482 of *Lecture Notes in Artificial Intelligence*, pages 398–411. Springer Verlag, 1991.

[11] Beatriz López and Enric Plaza. Case-based planning for medical diagnosis. In Z. Ras, editor, *Methodologies for Intelligent Systems*, volume 689 of *Lecture Notes in Artificial Intelligence*, pages 96–105. Springer Verlag, 1993.

[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] Enric Plaza. Reflection for analogy: Inference-level reflection in an architecture for analogical reasoning. In *Proc. IMSA'92 Workshop on Reflection and Metalevel Architectures*, pages 166–171, 1992.

[15] 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.

[16] Carles Sierra, Lluís Godo, Ramon López de Mántaras, and Mara Manzano. Descriptive Dynamic Logic and its application to reflective architectures. *Future Generation Computer Systems*, 1996. in this issue.

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

[18] J. Treur. On the use of reflection principles in modelling complex reasoning. *Int. J. Intelligent Systems*, 6:277–294, 1991.

[19] Bob Wielinga, Guss Schreiber, and Joost Breuker. KADS: A modelling approach to knowledge engineering. *Knowledge Adquisition*, 4(1):5–54, 1992. Special Issue 'The KADS approach to knowledge engineering'.

[20] 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.