Chapter 3

The Noos Approach

In this chapter we present incrementally the Noos representation language. The chapter is organized in six main sections: Section 3.1 presents the Noos modeling framework based on four knowledge categories: *domain knowledge*, *problem solving knowledge*, *episodic knowledge*, and *metalevel knowledge*. Section 3.2 describes the basic elements of the language such as descriptions, refinement, references, methods, and the basic inference. Section 3.3 presents the reflective capabilities of Noos introducing elements such as metalevels, tasks, reflective operations, reification, and reinstantiation. Section 3.4 describes *preferences*, a declarative mechanism for decision making about sets of alternatives. Section 3.5 presents the complete Noos inference engine. Finally, Section 3.6 summarizes the main features of the Noos language.

3.1 The Noos modeling framework

Knowledge-based problem solving is characterized by the intensive use of highly domain specific elements of knowledge. The purpose of knowledge modeling approaches is to describe this knowledge and how it is being used in a particular problem in an implementation independent way. Different knowledge modeling approaches have proposed different categories of knowledge elements and different abstractions to describe them (see Section 2.1).

We propose a model based on four knowledge categories: *domain knowledge*, *problem solving knowledge*, *episodic knowledge*, and *metalevel knowledge*.

Domain knowledge

The first knowledge category of the Noos framework is *domain knowledge*. Domain knowledge specifies a set of *concepts*, a set of *relations* among concepts, and problem data that are relevant for an application. Concepts and relations define the *domain ontology* of an application. For instance, Figure 3.1 shows part of the domain ontology defined in the diagnosis of car malfunctions application.

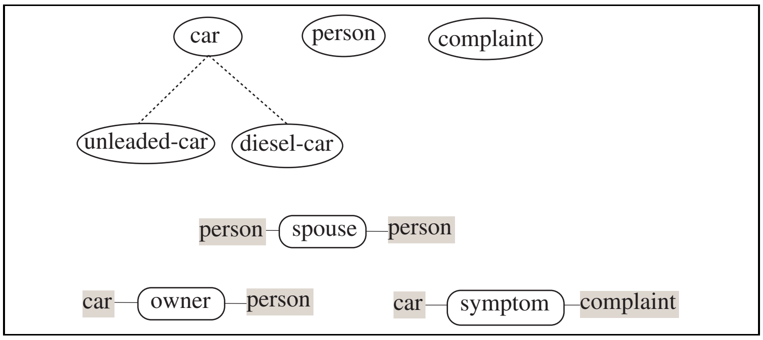


Figure 3.1. Part of the domain ontology of the diagnosis of car malfunctions application.

Problem data, described using the domain ontology, define specific situations (specific problems) that have to be solved. For instance, in the diagnosis of car malfunctions application, problem data represents specific car problems. Problem data constitutes part of the *episodic model* (see below).

Problem solving knowledge

Another category of the Noos framework is *problem solving knowledge*. Problems to be solved in a domain are modeled as *tasks*. For instance, following the previous example, the main task in the cars diagnosis domain is to establish malfunctions of cars. In our approach, a *method* models a way to solve problems. Methods can be elementary or can be decomposed into subtasks. These new (sub)tasks may be achieved by other methods. A method defines a specific combination of the results of the subtasks in order to solve the task it performs. For a given subtask there may be multiple alternative methods that may be capable of solving that subtask in different situations. This recursive decomposition of a task into subtasks by means of a method is called the task/method decomposition. For instance, Figure 3.2 shows a browser of the task/method decomposition of `general-diagnosis` method (following [Benjamins, 1993]). The `general-diagnosis` method is decomposed into three subtasks, namely `detect-complaint`, `generate-hypothesis`, and `discriminate-hypothesis`. For each subtask one or several alternative methods are specified—e.g. subtask `detect-complaint` has methods `ask-user`, `classify`, and `compare`.

Episodic knowledge

Problem solving in Noos is considered as the construction of an *episodic model*. In this sense the Noos approach to problem solving is close to that of CommonKADS [Wielinga et al., 1993] and to the TASK language

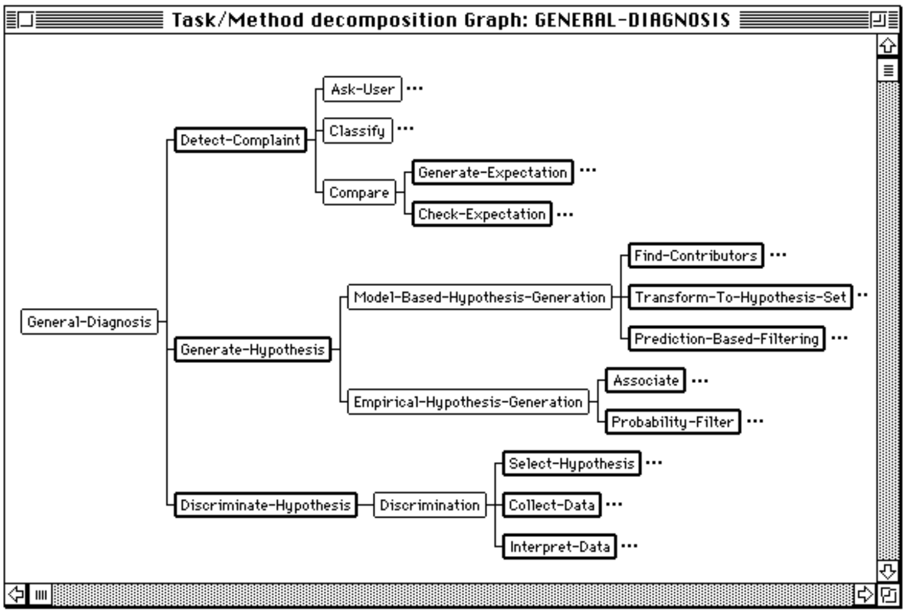


Figure 3.2. A browser of the task/method decomposition for a general diagnosis method. Methods are drown with thin boxes; tasks are drown with thick boxes; dots indicate not expanded terms.

[Pierret-Golbreich and Hugonnard, 1994]. Our view of "problem solving as modeling" is that problem solving is the process of constructing an episodic model. This model is obtained from transformations of problem data performed using problem solving knowledge. A clear and explicit separation between tasks, methods, and domain knowledge permits the dynamical link between a given problem, tasks, and methods as well as the dynamical choice of a method suited to achieve a task in that problem context: a 'task' uses a 'method' on a 'episode' (described using domain knowledge and problem data). An episodic model is the set of knowledge pieces used for solving a specific problem. Once a problem is solved Noos automatically memorizes (stores and indexes) the episodic model that has been built. Episodic memory (see Section 4.1) is the (accessible and retrievable) collection of the episodic models of the problems that a system has solved. The memorization of episodic models is a basic building block for integrating learning into a KM framework.

Metalevel knowledge

The last category of the Noos framework is *metalevel knowledge*. Metalevel (or reflective) knowledge is knowledge *about* domain knowledge, problem solving

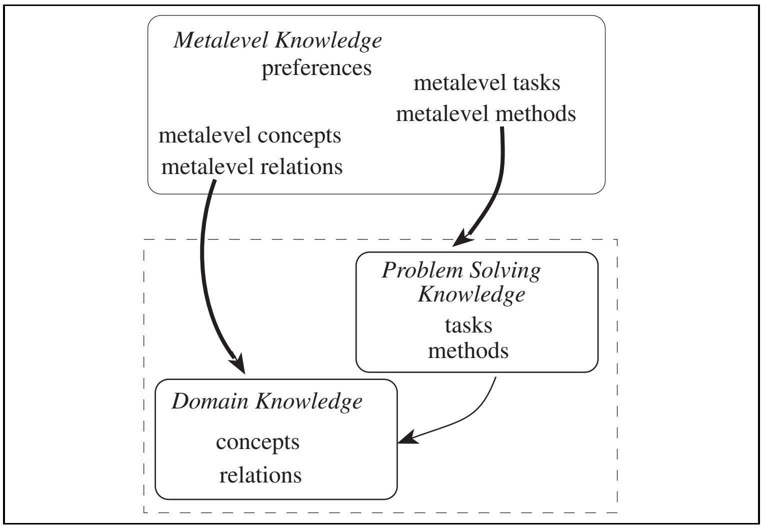


Figure 3.3. The Noos modeling framework.

knowledge, and episodic knowledge.

On the first hand, metalevel knowledge can have models about concepts, relations, tasks, and methods. These models are formed by *metalevel concepts*, *metalevel relations*, *metalevel tasks*, and *metalevel methods* (see Fig. 3.3). Moreover, metalevel knowledge includes *preferences* to model decision making about sets of alternatives present in domain knowledge and problem solving knowledge. For instance, metalevel knowledge can be used to model criteria for preferring some methods over other methods for a task in a specific situation. Metalevel concepts and metalevel relations can be used to build particular views of the domain knowledge defining more abstract models, such as causal models, that ease the usage of domain knowledge. An example of metalevel task is one that chooses—from a set of alternative methods—a specific method for a given task. An example of metalevel method is one that—for a specific situation—searches possible methods for a task, selects some methods as suitable alternatives, and finally ranks them using a set of preferences (see Section 3.3.1).

On the other hand, metalevel knowledge have models about how problems are solved in the system—or in other words, knowledge about episodic models. Knowledge about episodes models knowledge such as the method that has succeeded in achieving a specific task, the methods that have failed in achieving a specific task, the tasks that have been engaged in the evaluation of a method, and the preference orders built while choosing among alternatives. Metalevel knowledge about episodic knowledge is the basis for the integration of learning in Noos (see Chapter 4).

3.1.1 Related work

Our purpose in the design of the knowledge categories of Noos was to use a set of knowledge categories close to the knowledge modeling proposals such as KADS [Wielinga et al., 1993] and Components of Expertise [Steels, 1990]. We are interested in proposing a compatible approach in order to take advantage of their work. Specifically, the research on problem solving methods (PSM) [Benjamins, 1993] is able to help the design of problem solving methods in Noos. Moreover, our work on incorporating learning capabilities can benefit to other existing knowledge modeling proposals.

Nevertheless, there are some differences between Noos and the other proposals. The first difference is that since in our approach a PSM defines a way in which a task can be achieved, PSMs determine the subtask decomposition of tasks. In this sense we have joined tasks and methods as elements of problem solving knowledge. Another difference is that task specification and method selection knowledge, as are defined in KADS, is modeled as metalevel knowledge in Noos.

3.2 The Noos language

Noos is an object-centered representation language based on *feature terms*. Feature terms are record-like data structures embodying a collection of *features* that are a generalization of first order terms. Feature terms and its relation with other works are presented in Chapter 5. For now we will only introduce some intuitions about feature terms.

In first order terms the parameters of a predicate are identified by position. For instance, we can define a predicate `person` containing four parameters as follows

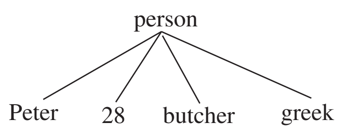
$$\texttt{person}(x_1, x_2, x_3, x_4)$$

with the implicit assumption that the first argument of the constructor `person` carries the "feature" `name`, the second argument carries the "feature" `age`, the third argument carries the "feature" `profession`, and the fourth argument carries the "feature" `nationality`.

Using this predicate we can describe a person whose name is `Peter`, whose age is 28, whose profession is `butcher`, and whose nationality is `greek` writing the following term:

$$\texttt{person}(\texttt{Peter}, 28, \texttt{butcher}, \texttt{greek})$$

A first order term is formally described as a tree with a fixed tree traversal order. For instance, the previous term is represented as the following tree with a left-to-right ordering:



Let us suppose now that we want to describe a person whose name is `Janet` and whose profession is `engineer` ignoring the age and the nationality. This kind of representation results inappropriate for representing incomplete information since it requires to specify all the arguments. Another limitation presented in this representation is the support for *extensibility* (we cannot add a new "feature" like spouse that would involve a fifth parameter). The fixed number of and the positional meaning of parameters is the cause of these shortcomings—and the justification for research performed on feature terms [Aït-Kaci and Podelski, 1993].

The intuition behind a feature term is that of providing a way to construct terms embodying *partial information* and amenable to extension. The proposal of feature terms is that these goals can be achieved by building terms with parameters identified by name (regardless of order or position) and with no fixed number of parameters. For instance, the previous example is specified as a feature term as follows:

$$\texttt{person}[\texttt{name} \doteq \texttt{Peter}\ \ \texttt{age} \doteq 28\ \ \texttt{profession} \doteq \texttt{butcher}\ \ \texttt{nationality} \doteq \texttt{greek}]$$

saying that the feature term has sort `person`, its feature `name` is `Peter`, its feature `age` is 28, its feature `profession` is `butcher`, and its feature `nationality` is `greek`.

The identification of parameters using names instead of position allows to represent partial knowledge. Thus, we can describe a person whose name is `Janet` and whose profession is `engineer`, ignoring other features, as follows:

$$\texttt{person}[\texttt{name} \doteq \texttt{Janet}\quad \texttt{profession} \doteq \texttt{engineer}]$$

More formally, while first order terms can be described by trees with an implicit ordering, feature terms can be seen as a generalization of them and can be described by labeled graphs where nodes are labeled with sorts and edges are labeled with named parameters (called *features*). For instance, the two previous descriptions are represented as labeled graphs as follows:

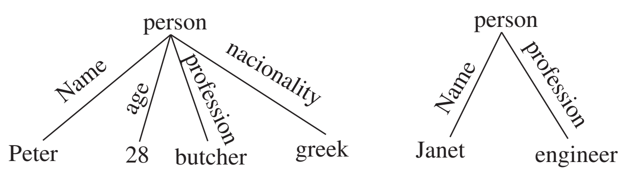


Figure 3.4. Labeled graph representation of two feature terms.

There are two types of feature terms in Noos: *constant feature terms* and *evaluable feature terms* (Feature terms are described formally in Chapter 5). Domain knowledge, as defined by the Noos model, is mapped to the Noos language as constant feature terms. A feature term F representing a concept C clusters together (as features) the relations in which C is involved. Methods are mapped to the Noos language as evaluable feature terms. Numbers, strings and symbols are considered as predefined constant feature terms without features.

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

After this introduction about feature terms, we will present in the next section *descriptions*, the syntax Noos uses for constructing feature terms. Following sections describe the different elements that configure the Noos language: the second section is dedicated to explain the refinement capabilities provided by code reuse and subtyping. The following section presents the use of references in the language. The next section explains the way problem solving knowledge—tasks and methods—is represented in the language. The last section presents the basic inference process of Noos.

3.2.1 Descriptions

Descriptions are the syntax Noos uses for constructing feature terms. The description syntax is based on lists (like Lisp) starting with token `define`, followed by a *name* (identifier), and some *features*. A *feature* is a pair of *feature name* and *feature value*. A feature value can be simply the name of a feature term described elsewhere. For instance, the following two descriptions[1]:

```
(define Peter
  (age 28)
  (wife Mary))

(define Mary
  (age 27)
  (husband Peter))
```

construct two feature terms; the first feature term with name `Peter` containing two features: `age` and `wife` with corresponding feature values 28 and `Mary`; The second feature term with name `Mary` containing two features: `age` and `husband` with corresponding feature values 27 and `Peter`.

Features describe direct labeled relations among terms. For instance, in the first previous description a direct relation labeled as `wife` is specified from `Peter` to `Mary`.

Since a name in a feature value denotes a feature term, a symbol is described with the quote operator for distinguishing to references to feature terms. Strings are described using double quotes at the beginning and at the end. For instance, the following non-sense description

[1]In the Noos language, like Lisp, there is no difference in the case of letters used—Noos is a case insensitive language. For instance, `Car`, `car`, and `CAR` are all valid and equivalent.

```
(define Peter-variants
  (name-reference Peter)
  (symbol 'Peter)
  (string "Peter"))
```

defines, in turn, a name of the feature term `Peter`, the symbol `'Peter`, and the string `"Peter"`.

Noos allows features where the feature value is a set. The syntax used for feature set values is the enumeration of names. For instance, the previous description of `Peter` can be extended incorporating the description of his children as the set composed by `Sara`, `Paul`, and `Shirley`.

```
(define Peter
  (age 28)
  (wife Mary)
  (children Sara Paul Shirley))
```

3.2.2 Refinement

The notion of *refinement* is introduced in the Noos language as a methodology to define a feature term from another existing feature term. The feature term that is reused is called the *constituent*. Refinement involves two distinct aspects: *code reuse* and *subtyping*, that will be explained below.

Syntactically, a description by refinement is a list composed of the `define` token, a *constituent*, a *name* and a *body* embodying a collection of features. The syntax of a description by refinement is the following:

Named Description `(define (`*constituent name*`)` *body*`)`

where *name* and *body* have the same meaning as defined before and *constituent* is the name of the reused feature term. The *name* is optional and when it is not given we say that an *anonymous* feature term is defined:

Anonymous Description `(define (`*constituent*`)` *body*`)`

Figure 3.5 describes the basic Noos syntax that will be explained in following sections (The complete Noos syntax can be found in Appendix C).

Anonymous feature terms can be also defined as feature values. For instance, we may define the concept `person` as a feature term whose `father` is also a person of sex `male` and whose `mother` is another person of sex `female` as follows:

```
(define Person
  (father (define (person)
            (sex male))
  (mother (define (person)
            (sex female))))
```

We say that descriptions if features `father` and `mother` are *subdescriptions* of `Person` and that `Person` is the *root* description.

description	::=	single-description
	|	named-description
	|	anonymous-description
	|	metalevel-description
	|	set-description
single-description	::=	`(define` *name* body`)`
named-description	::=	`(define (`*constituent* [`:id`] *name*`)` body`)`
anonymous-description	::=	`(define (`*constituent*`)` body`)`
metalevel-description	::=	`(define (`*constituent* `(meta+ of` *name*`))` body`)`
set-description	::=	`(define (set` *name*`)` value+`)`
body	::=	feature-description*
feature-description	::=	`(`*feature-name value**`)`
	|	`(`*feature-name* path-reference`)`
	|	`((`*feature-name* value+`))`
value	::=	*name* | anonymous-description
path-reference	::=	`(>>` *feature-name** [`of` *name*]`)`

Figure 3.5. This figure shows a subset of Noos syntax used for the definition of descriptions in BNF notation. Remark that in *feature-description* double parenthesis are used to define methods. Typewriter font words are predefined terminal symbols that are part of the language, italic words are user-defined identifiers and ::=, |,[ ], + and * are part of the BNF formalism.

Code reuse

Using refinement a new feature term can be constructed reusing another existing feature term. Specifically, a new feature term N defined as a refinement of another feature term E as `(define (`E N`)` *body*`)` includes (reuses) all the features defined in E that are not redefined in *body*. In other words, N extends E with features defined in *body*. For instance, we may define the concept of a `citizen`, with two features `lives-at` and `pays`, as follows

```
(define citizen
  (lives-at region)
  (pays taxes))
```

Then, we model the citizens of a given region by a refinement of `citizen` that overrides the feature value of the `lives-at` feature with the name of that specific region. In addition, we may incorporate new features like the `language` spoken by citizens of that region. For instance, we may define a `bagenc` as a citizen that lives at `Bages` (a beautiful region of Catalonia) and that `speaks` Catalan and Spanish.

```
(define (citizen bagenc)
  (lives-at Bages)
  (speaks catalan spanish))
```

Finally, a specific citizen `Pep` can be defined describing the languages he speaks by overriding the `speaks` feature and incorporating a new feature such as his interests:

```
(define (bagenc Pep)
  (speaks catalan english)
  (likes climbing))
```

Recall that descriptions are syntax for building feature terms, so the feature term constructed for `Pep` in fact includes the feature `pays` from `citizen` description, the feature `lives-at` from `bagenc` description, and features `speaks` and `likes` from `Pep` description. The graph representation of the feature term `Pep` is given in the browser of Figure 3.6.

Figure 3.7 shows another example of definition by refinement. First, the `car` description is defined containing the common knowledge about cars. Then, specific models of cars can be defined by refinements of `car`. Finally, specific cars can be described as refinements of models of cars. Thus, all the features defined in `car` and `Ibiza-car` that are not redefined in the car that `Peter drives` are included in this new feature term.

Sorts

Noos provides an initial set of sorts with an order relation among them. There is a top sort called `any`. `Any` represents the minimum information and all the other sorts are more specific than `any` (for each sort `S` we have that `any` $\leq$ `S`). Predefined sorts are for instance (see Appendix A for a description of the complete Noos predefined sort hierarchy):

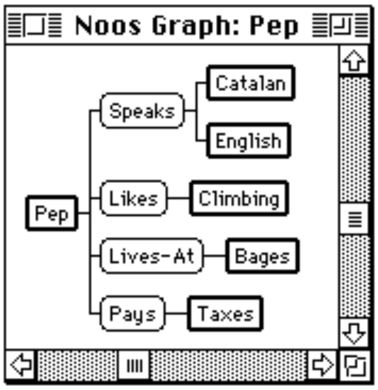


Figure 3.6. A Noos browser visualizing the labeled graph representation of `Pep`. Feature names are represented as thin boxes. Note that feature terms are rooted labeled graphs. The root node in this example is node `Pep`.

- all numbers and the sort `number`, with the order relations `number` $\leq n$ for all numbers n,
- all strings and the sort `string`, with the order relations `string` $\leq s$ for all strings s,
- all symbols and the sort `symbol`, with the order relations `symbol` $\leq s$ for all symbols s,
- sorts `boolean`, `true`, and `false` with the order relations `boolean` $\leq$ `true` and `boolean` $\leq$ `false`,
- sorts `set` and `empty-set` with the order relation `set` $\leq$ `empty-set`.

From this set of initial sorts new sorts can be defined, using refinement, for specifying the sort hierarchy for a given domain.

Subtyping

Using refinement we are specifying the sort of the feature term and we are also defining a sort hierarchy for a given domain. On the one hand, names of feature terms are interpreted as sorts. This means that when we define a named term, we are also defining a sort with the same name. On the other hand, the notion of refinement involves the construction of an order relation $\leq$ among the sorts.

Specifically, a description such as `(define (X Y) body)` defines a new sort `Y` with an order relation `X` $\leq$ `Y` with the existing sort `X`. Moreover, the sort of the new feature term being constructed is `Y`. For instance, the definition of `Ibiza-car` as a refinement of `car` involves (1) the definition of a new sort

```
(define Car
  (owner (define (person)))
  (gas-level-in-tank gas-level)
  (gas-gauge-reading (>> gas-level-in-tank))
  ((empty-level?  (define (Identity?)
                    (item1 empty)
                    (item2 (>> gas-gauge-reading)))))
  (price (>> price model)))

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

(define (car-model Ibiza)
  (manufacturer Seat)
  (price 12000))

(define (Person :id Peter)
  (age 28)
  (spouse Mary)
  (drives (define (Ibiza-Car)
             (owner (>> spouse))
             (symptom does-not-start)
             (gas-level-in-tank full))))
```

Figure 3.7. `Car`, `Ibiza-car`, `Ibiza`, and `Peter` are defined using Noos descriptions; features `owner`, `gas-level-in-tank`, `gas-gauge-reading`, `empty-level?` and `price` are defined for the `Car` description; the feature `model` is defined for `Ibiza-car`; features `manufacturer` and `price` are defined for the `Ibiza` car model; features `age`, `spouse`, and `drives` for `Peter`; finally, `owner`, `symptom` and `gas-level-in-tank` features are defined in the (sub)description of the car that `Peter` drives. For brevity, some descriptions like `car-model` or `person` are not included. The Noos basic syntax is summarized in Figure 3.5.

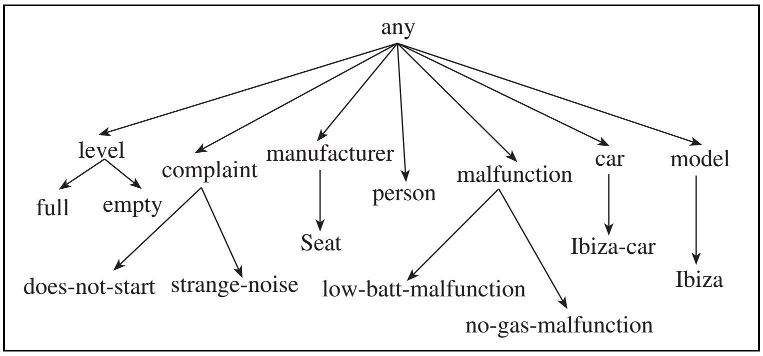


Figure 3.8. The Noos sort hierarchy used in the example of Fig. 3.7 (the diagnosis of car malfunctions domain).

`Ibiza-car` and, (2) an order relation: `car` $\leq$ `Ibiza-car` (see Figure 3.8 for the Noos sort hierarchy relevant to the diagnosis of car malfunctions domain).

When we define a feature term using an anonymous description we are not defining a new sort. An anonymous description such as `(define (X) body)` defines a feature term that has as sort `X`.

The problem with anonymous feature terms is that, since they have no name, they cannot be referred to by name. When we want to define a feature term without introducing a new sort, but we want to have an identifier in order to refer to it, Noos provides an extended syntax for named descriptions using the `:id` token as follows:

Named Description `(define (`*constituent* [`:id`] *name*`)` *body*`)`

When we use the optional token `:id`, we are defining a feature term with name *name* without introducing a new sort. For instance, for the previous definition of a specific citizen `Pep`, it was more appropriate the use of the *:id* token as follows:

```
(define (bagenc :id Pep)
  (speaks catalan english)
  (likes climbing))
```

since we are using the name `Pep` as an identifier and not as a sort.

Named feature terms defined with the `:id` token cannot be refined.

The intuition about the order among sorts is that it specifies an informational ordering: given two sorts `X` and `Y`, the order `X` $\leq$ `Y` is interpreted as `Y` contains all the information contained in `X` plus more information. In other words, `Y` is more specific than `X` (see a detailed description in Chapter 5).

This sort hierarchy is the basis to model an order relation among feature terms. We call this order relation among feature terms *subsumption* (see Section 5.6).

All the feature terms constructed in Noos are in fact definitions by refinement. Specifically, examples of descriptions such as `Peter` and `Mary` given in the previous section are also considered definitions by refinement from a feature term without features whose sort is the top sort `any`. When a description is constructed directly as a refinement of this top feature term, Noos provides the compact syntax

Single description `(define` *name body*`)`

that is equivalent to

`(define (any` *name*`)` *body*`)`

Named sets

Another kind of descriptions provided in Noos are *named sets*. Named sets allows to group descriptions in an identifier and refer to them using only the name. The syntax of named sets is the following,

`(define (set` *setname*`)` $name_1 \cdots name_n$`)`

where each $name_i$ is a name of a feature term described elsewhere. For instance, we can group our favorite colors in the named set `my-colors` and refer them elsewhere

```
(define (set My-colors)
  yellow green blue)
```

Concluding remarks about refinement

We have presented the notion of refinement as a crucial methodology of the Noos language for constructing feature terms that involves two distinct aspects: (1) code reuse (the construction of a feature term by reusing another feature term) and, (2) subtyping (the definition of the sort hierarchy). As a summary of this section we will describe briefly the different aspects involved in each kind of definition by refinement previously presented:

- A named description such as `(define (X Y) Z)` involves:
  - the definition of a new sort `Y`,
  - the definition of an order relation $X \leq Y$ between sorts `X` and `Y`, and
  - the definition of a feature term with name `Y`, with sort `Y`, with the features defined in `Z`, and including all the features defined in `X` that are not redefined in `Z`.
- An anonymous description such as `(define (X) Z)` involves:
  - the definition of a feature term with sort `X`, with the features defined in `Z`, and including all the features defined in `X` that are not redefined in `Z`.

- A named description such as `(define (X :id Y) Z)` involves:
  - the definition of a feature term with name `Y`, with sort `X`, with the features defined in `Z`, and including all the features defined in `X` that are not redefined in `Z`.

3.2.3 References

There are two forms of references in Noos: *name references* and *path references.* We already have used name references—for instance, the feature value of feature `model` of `Ibiza-car` in Figure 3.7 is defined using a name reference to `Ibiza`. However anonymous feature terms cannot be referred to by named references, anonymous feature terms are referred to using path references. There are two kinds of path references: *absolute* and *relative* path references.

Absolute path references

An absolute path reference is a list that starts with the `>>` token, followed by a sequence of feature names, then the `of` token, and finally the name of a named feature term. Following an example from Figure 3.7, an absolute path reference to the `symptom` of the car that `Peter drives` (which is `does-not-start`) is written:

```
(>> symptom drives of Peter)                (3.1)
```

It is clear that this reference is in fact the concatenation of two references: The reference to the car that `Peter drives`, which is an `Ibiza-car`, and the reference to its `symptom`. These two concatenated references can also be written as:

```
(>> symptom of (>> drives of Peter))        (3.2)
```

In fact, expression (3.1) is just shorter syntax for expression (3.2). Considering that the path reference `(>> drives of Peter)` refers to a feature term with print name `<Ibiza-car-33>` (print names of feature terms are written in Noos inside angles) expression (3.2) is equivalent to the path reference

```
(>> symptom of <Ibiza-car-33>)              (3.3)
```

Finally, since the value of the `symptom` feature of `<Ibiza-car-33>` is the `does-not-start` complaint, expression (3.3) is equivalent to the name reference `does-not-start`.

Relative path references

A relative path reference elides the name of a feature term and specifies only a sequence of feature names, e. g. `(>> price model)`. A relative path reference is bound to a specific description by the rules of scope and refinement.

Scope

Scope in Noos is *lexical*; that is to say, a relative reference is determined *by the text in which it appears*. Specifically, a relative path reference is bound to the *root* of the description in which it appears—the outmost `define` in the text where it occurs. For instance, considering the following description of `Peter` driving a specific model of `Ibiza-car` (let us call it `<Ibiza-car-33>`):

```
(define (person :id Peter)
  (age 28)
  (spouse Mary)
  (drives (define (Ibiza-Car)
            (owner (>> spouse))
            (symptom does-not-start)
            (gas-level-in-tank full))))
```

the relative path reference `(>> spouse)` in the feature `owner` of the `<Ibiza-car-33>` refers to `Peter` (and not to `<Ibiza-car-33>`) since `Peter` is the root of the description.

In Figure 3.7 root descriptions are `car`, `Ibiza-car`, `Ibiza`, and `Peter`, so relative path references appearing in those descriptions are bound to those roots.

Scope plus refinement

A relative path reference defined in a description `D` and incorporated by refinement into a new description `D'` is bound in the scope of `D'` relatively to the root description where `D` was *textually* defined.

For instance, feature `price` is defined in `car` by the relative path reference `(>> price model)` as follows:

```
(define Car
  (price (>> price model)))
```

Thus, the relative path references occurring in `car` description (`price` feature in this example) are always relative to the appropriate `car` in whatever refinement of `car`, regardless of whether that car is a subdescription of another description.

Then, since in the previous example `<Ibiza-car-33>` is defined by refinement of `Ibiza-car`—and `Ibiza-car` is defined by refinement of `car`—feature `price` is incorporated to `<Ibiza-car-33>` bound to `<Ibiza-car-33>` (and *not* to `Peter`).

The Noos scope is formally explained in Chapter 5.

Null path

The sequence of feature names in a path reference can be empty. A null path `(>>)` is a relative path reference that denotes directly the root of the description in which it textually appears. For instance, in the following description of `Janet`

```
(define (person Jane)        (define (person Arthur)
  (brothers Adam Arthur)       (wife Lucy))
  (sisters Abigail Alison))
                             (define (person Linda)
(define (person Abigail)       (brothers Clement Charles))
  (husband Bob))
                             (define (person Lucy)
(define (person Alison)        (brothers David Douglas))
  (husband Bart))

(define (person Adam)
  (wife Linda))
```

Figure 3.9. Family relations example.

```
(define (person :id Janet)
  (drives (define (Car)
            (owner (>>))
            (symptom does-not-start)
            (gas-level-in-tank empty))))
```

the null path reference in the `owner` feature of the `car` subterm would refer to `Janet`.

Reference over sets

Since feature values can be sets, path references have to deal with feature values that are sets. For instance, since a feature like `brothers` may have as value a set of feature terms, a path reference like `(>> wife brothers of Jane)` should be understood as referring to the sisters-in-law of `Jane`. Using the family relations described in Figure 3.9, since the brothers of `Jane` are (the set of) `Adam` and `Arthur`, the question is the meaning of `(>> wife of <Set of Adam Arthur>)`. A path reference over a set is interpreted as the set of element-wise path references. The element-wise definition of a reference over sets indicates that the reference `(>> wife of <Set of Adam Arthur>)` is interpreted as the set of references `(>> wife of Adam)` and `(>> wife of Arthur)` —that is to say the set of `Linda` and `Lucy` (see Figure 3.9).

It is also important to notice that references over sets produce *flat sets*. In other words, the result of a reference over a set is a set containing all the results and is *not* a set of sets of results. For instance, the path reference

```
(>> brothers wife brothers of Jane)
```

yields the result

```
<Set of David Charles Clement Douglas>
```

which is the set of objects that are the brothers of the wives of the brothers of `Jane`. Notice, in particular, that Noos will *not* return as result from a reference over a set something like

```
<Set of <Set of David Douglas> <Set of Charles Clement> >
```

which is a set of sets of values.

Path equality

A property of feature terms is that subterms of a feature term are univocally determined by a path leading to a subterm from the root. For instance, we may define `Edward` as follows:

```
(define (Person Edward)
  (lives-at (define (address)
              (city (define (city)
                      (name Manchester)
                      (in England)))))
  (daughter (define (child)
              (lives-at (>> lives-at)))))
```

where the `address` subterm is determined by the path

```
(>> lives-at of Edward)
```

the `city` subterm is determined by the path

```
(>> city lives-at of Edward)
```

and the `child` subterm is determined by the path

```
(>> daughter of Edward)
```

Another property of feature terms is that of *path equality*. Since any feature value can be determined by a path from the root feature term, given a feature `F1`, determined by a path `P1` leading from the root to `F1`, with feature value a path `P2`, `P2` determines a path equality with `P1`.

For instance, the path reference `(>> lives-at)` in feature `lives-at` in `Edward`'s child subterm specifies the following path equality:

```
(>> lives-at daughter of Edward) = (>> lives-at of Edward)
```

saying that the daughter of `Edward` lives at the same place as her father (`Edward`).

3.2.4 Methods

Methods are represented in Noos as evaluable feature terms and are constructed also by descriptions. The features of a method description have a different interpretation from that of descriptions of concepts: the set of features defined in a method description is interpreted either as a reference to some knowledge

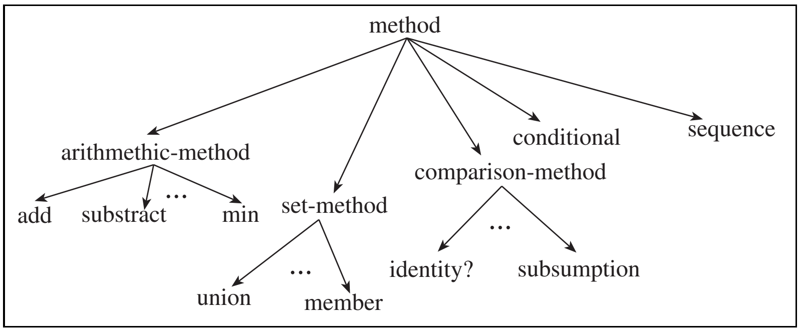


Figure 3.10. A partial expanded tree of the Noos built-in method hierarchy.

source required by the method, or as a subtask required to be accomplished by the method.

Intuitively, a method can be interpreted as a function the parameters of which are passed by name (the required feature names) instead of by position. A method reduces to a value as result of being evaluated. The value can be any feature term—including a method.

A second important aspect of a method is the specific way in which it combines the values of its features—i.e. the knowledge of sources and the results of its subtasks. Noos provides an initial set of existing methods, called *built-in methods*, that perform specific combinations of feature values. New methods can be defined by refinement of built-ins.

We will first present Noos built-in methods and then, how new methods are defined in Noos.

Built-in methods

Noos provides an initial set of built-in methods that perform specific combinations of knowledge sources and results of the subtasks (see Figure 3.10 for a partial description of Noos built-in method hierarchy).

For instance, the conditional function in Lisp is an expression (`if` x y z) where x is an expression returning a boolean, y is the expression evaluated when condition is true and z is the expression evaluated when condition is false. Clearly the role of each variable or subexpression is given by being in position 1, 2, or 3 of the parameter list. The conditional method in Noos is a built-in called `conditional`, and the three roles (that we call subtasks) are given by features with name `condition`, `result`, and `otherwise`. The `conditional` method performs first the subtask `condition` and depending on its result being `true` or `false`[2] either the `result` subtask or the `otherwise` subtask is performed and its result is the value yielded by conditional method.

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

Notice that, since the evaluation ordering of the subtasks of a method is performed taking into account the subtask names, subtasks of a method can be defined in any order. For instance, `conditional` can be written defining first the `result` subtask, then the `otherwise` subtask, and finally the `condition` subtask and the evaluation ordering will be the same.

Examples of Noos built-in methods are arithmetic operations, set operations, logic operations, operations for comparing feature terms, and other basic constructs such as conditional or sequencing (see Appendix D for a detailed explanation of Noos built-in methods). Each built-in method has a set of built-in *required* features. For instance, built-in method `Identity?` is a comparison method that compares two feature terms passed in features `item1` and `item2` returning `true` when are the same and `false` otherwise (`identity?` method works like `eq` predicate of Lisp).

Definition of methods

Methods are defined by refinement from built-in methods or other already defined methods. In order to illustrate the definition by refinement of methods in Noos we will introduce an example: a `causal-explanation` method to be used in the context of causal reasoning. A `causal-explanation` is a method with two subtasks: the first subtask checks whether a given `cause` occurs in a target problem, and if so the `effect` subtask is performed—and otherwise it fails (the notion of failure is described in Section 3.5). This method can be defined as a refinement of the built-in `conditional` method where features `condition` and `result` (the built-in required features defined for `conditional`) are refined as references to `effect` and `cause` feature values respectively.

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

Note that `cause` and `effect` are unspecified—since they are parameters to be passed in particular causal-explanations. It is not required to write these feature names in the description, but as a matter of style it helps in clarity to write the feature names referred to in a description even if they will not be specified until further refinements are made. Also note that the `otherwise` subtask of `conditional` is unspecified, thus when the `cause` subtask (equivalent to `condition` subtask) is not the case, the method will fail since there is no way to achieve the `otherwise` subtask. Next, more specific causal explanations can be defined through refinements of `causal-explanation`, as for instance:

```
(define (causal-explanation Wet-Causal-Expl)
  (location place)
  (cause (>> recent-rain?  location))
  (effect wet))
```

`Wet-Causal-Expl` has a new parameter, `location` and determines that such a location is `wet` whenever the feature `recent-rain?` of that location is `true`. This is still a generic (or parametric) method, although specialized for a given task. When we refine `Wet-Causal-Expl` method giving a specific place value for `location` feature, we will have a *closed* method.

We say that a method is *closed* when all the required features (references and subtasks) are specified (see section 5.9.1). In other words, a closed method is amenable to be evaluated and reduced to a value. For instance, if we refine `Wet-Causal-Expl` specifying a specific location as follows,

```
(define (Wet-Causal-Expl)
  (location (define (place)
              (recent-rain?  true))))
```

we are describing a closed method amenable to be evaluated and return a result (returning `wet` as result in this example).

Summarizing, first we have built the `Causal-Explanation` method by refining the `conditional` built-in method with two new named parameters called `cause` and `effect`. Then, we have defined `Wet-Causal-Expl` method by refinement of `Causal-Explanation` specifying parameters `cause` and `effect` and introducing a new parameter named `location`. Finally, we have defined a closed method specifying a specific location. This last method can be evaluated.

Methods in features

A closed method can be incorporated into a feature to infer its value. Specifically, introducing a closed method we describe a feature value by means of an inference instead of a constant or a path reference. In order to syntactically distinguish in a feature of a description a reference to a value from a method to infer a value, a method is indicated by a double parenthesis. That is to say, a feature with a method is written as follows:

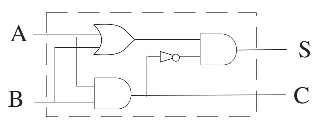
(define (*constituent name*)
  ((*feature-name closed-method*)))

For instance, in Figure 3.7 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.

Using another example, in the domain of digital logic circuits (see Figure 3.11) a `half-adder` component can be represented as a term with two input wires (represented by features `A` and `B`), and two output wires (represented by the features `S` and `C`). The value of feature `S` is defined using a composition of closed methods (`conjunction`, `disjunction`, and `not`) and will become `1` (`true`) whenever only one of `A` and `B` is `1`. The value of feature `C` is defined using another conjunction closed method and will become `1` whenever `A` and `B` are both `1`.

Next, we can define a `full-adder` component composed of two half-adders (represented by features `H1` and `H2`). The `full-adder` is the basic circuit for

```
(define Half-adder
  (A )
  (B )
  ((S (define (conjunction)
        ((item1 (define (disjunction)
                  (item1 (>> A))
                  (item2 (>> B)))))
        (item2 (define (not)
                 (item (>> C)))))))
  ((C (define (conjunction)
        (item1 (>> A))
        (item2 (>> B))))))
```




```
(define Full-adder
  (A )
  (B )
  (C-in )
  (H1 (define (half-adder)
        (A (>> B))
        (B (>> S H2))))
  (H2 (define (half-adder)
        (A (>> B))
        (B (>> C-in))))
  (Sum (>> S H1))
  ((C-out define (disjunction)
           (item1 (>> C H1))
           (item2 (>> C H2)))))
```

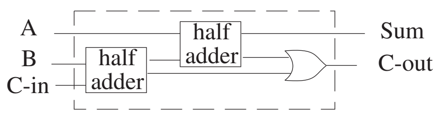


Figure 3.11. Definition of an adder circuit.

adding two binary numbers. A `full-adder` is represented as a term with three input wires and two output wires (represented as features): features `A` and `B` hold the bits at corresponding positions in the two numbers to be added. Feature `C-in` is the carry bit from the addition one place to the right. The value of feature `Sum` will hold the sum bit in the corresponding position. The value of feature `C-out` will hold the carry bit to be propagated to the left.

Concluding remarks

We have shown how new methods can be defined as a combination of other methods. That is to say, a new method is specified (1) by a refinement of another method that determines how the results of subtasks are combined, (2) by specifying which methods are used in each subtask, and (3) by specifying a set of named parameters.

Next, we can define closed methods, by specifying the required parameters of methods. A closed method can be incorporated into a feature to infer its value.

For instance, using the task/method decomposition for general diagnosis described in [Benjamins, 1993], a specific configuration of a model based diagnosis method can be defined as follows[3]:

```
(define (Sequence Model-based-diagnosis)
  (device faulty-device)
  ((detect-complaint (define (ask-user)
                       (source (>> device)))))
  ((generate-hypothesis (define (model-based-hypothesis-generation)
                          (device (>> device))
                          (symptom (>> detect-complaint)))))
  ((discriminate-hypothesis (define (discrimination)
                              (device (>> device))
                              (hypothesis (>> generate-hypothesis))))))
```

where the `Model-based-diagnosis` method is defined by refinement of the `sequence` built-in method, with a named parameter `device`, and decomposed into three subtasks, namely `detect-complaint`, `generate-hypothesis`, and `discriminate-hypothesis`. For each subtask one method is specified: The `detect-complaint` task is performed by the `ask-user` method that requests to the user for determining complaint symptoms; the `generate-hypothesis` task is performed by the `model-based-hypothesis-generation` method that uses a domain model to generate hypotheses that explain the set of initial observations from the `device`; and `discriminate-hypothesis` task is performed by the `discrimination` method that determines if an hypothesis generated by previous task has to be discarded.

[3]see Figure 3.2 in page 29 for a browser of a task/method decomposition for general diagnosis.

3.2.5 Inference

After describing the basic components of the Noos language, we are ready to introduce the main intuitions on the basic inference process of Noos. The whole Noos inference process involving metalevel inference and reasoning about preferences, however, is described in Section 3.5.

Inference in Noos involves three processes: the inference of feature values, the reduction of path references, and the evaluation of closed methods. Since a feature value can be defined either as a constant value, as a path reference, or by means of the evaluation of a closed method, the inference of a feature value can involve, in turn, the reduction of a path reference and the evaluation of a closed method.

A main characteristic of inference in Noos is that it is on demand (also called *lazy* inference). On demand inference means that no inference is performed until it is required.

Inference starts when the user poses a query to the system by means of a *query expression*. There are two kinds of query expressions: *path references* and *eval expressions*.

Path references

Path references can be used as query expressions. Specifically, when a query expression `(>> F of D)` is posed to the system and the feature value for feature `F` of `D` is unknown, the task of inferring the corresponding feature value is *engaged* in Noos.

Tasks engaged by query expressions are called *problem tasks*. A problem task `F(D)` engages the inference to determine the feature value for feature `F` of feature term `D`. For instance, the following query expression:

```
(>> diagnosis of Peters-car)
```

engages the problem task `diagnosis(Peters-car)` for determining the feature value of feature `diagnosis` of `Peters-car`.

Eval expressions

Another way to specify a query expression is by requesting the evaluation of a closed method using the following syntax:

```
(noos-eval M)
```

that engages the evaluation of the closed method `M`.

For instance, the following query expression:

```
(noos-eval (define (Wet-Causal-Expl) (location my-home)))
```

engages the evaluation of a closed method defined by refinement of the `Wet-Causal-Expl` method where a specific place value `my-home` for the `location` feature is given (see Section 3.2.4 for the definition of the `Wet-Causal-Expl` method).

The inference engine

From the inference process engaged by a query expression, the Noos inference engine can be described using three basic processes: the *Engage-Task* process, involving the inference of feature values; the *Reduce-Path-Reference* process, involving the reduction of a path reference; and the *Noos-Eval* process, involving the evaluation of a method. These three processes will be presently explained in turn.

Engage-Task

The goal of the *Engage-Task* process is to infer the value of a feature. *Engage-Task* is engaged for solving a problem task. *Engage-Task* process involves four different actions according to the different feature value specifications as follows:

Given a task `F(D)` engaged for determining the value of a feature `F` of a feature term `D`,

1. When there is a constant feature value `C` defined for feature `F` of `D`, the task is achieved yielding `C`.

2. When there is a path reference `R` defined for feature `F` of `D`, the process *Reduce-Path-Reference* is engaged for path reference `R`. The task is achieved if the reduction process succeeds, yielding the feature term to which the path reference `R` reduces. Otherwise *Engage-Task* fails.

3. When there is a closed method `M` defined for feature `F` of `D`, the *Noos-Eval* process is engaged for method `M`. The task is achieved if the evaluation of method `M` succeeds, yielding the value inferred in that evaluation. Otherwise *Engage-Task* fails.

4. When neither a path reference nor a method is defined for feature `F` of `D`, an *impasse* occurs. In this situation the control of the inference is passed to the metalevel. Metalevel inference is explained in Section 3.5, so we will ignore this case until section Section 3.5.

Once the task of inferring the value for a feature `F` of a term `D` is achieved, the inferred value is automatically cached in the feature `F`. Caching mechanism allows Noos to answer quickly future demands of previously inferred feature values (see Section 3.5).

Reduce-Path-Reference

The goal of the *Reduce-Path-Reference* process is to infer the feature term referenced by a path reference. As we have shown in Section 3.2.3, a path reference like `(>> F1 F2 of D)` is in fact a concatenation of two references that can be defined in the following equivalent syntax:

```
(>> F1 of (>> F2 of D))
```

We will call this form a *canonical* path reference. The *Reduce-Path-Reference* process will be described considering only canonical path references.

Given a canonical path reference `R`, the *Reduce-Path-Reference* process can be described recursively as follows:

1. when the path reference `R` is of the form `(>> F of D)`, being `F` a feature name and `D` a name reference, the *engage-task* process is engaged for solving the task `F(D)`. If *engage-task* fails, then the *Reduce-Path-Reference* process fails. Otherwise *Reduce-Path-Reference* yields the value inferred by task `F(D)`.

2. when the path reference `R` is a relative path reference of the form `(>> F)`, the system determines the corresponding absolute path `(>> F of D)` following the rules of scope and refinement (see Section 3.2.2). Then, the *engage-task* process is engaged for solving the task `F(D)`.

3. when the path reference `R` is of the form `(>> F of R')`, being `R'` another path reference, then the *Reduce-Path-Reference* process recursively reduces `R'`. If we call `D` the feature term yielded in the reduction of `R'`, then the *engage-task* process is engaged for solving the task `F(D)`.

Noos-Eval

The *Noos-Eval* process for a method `M` first engages a task for each required feature of `M` and then performs a specific combination of the results of the subtasks according to the built-in method it is a refinement of. The *Noos-Eval* process is engaged for solving an eval expression.

Specifically, given a method `M` that is a refinement of a built-in method `B` with required features F_1, F_2, $\cdots$, F_n,

1. tasks $F_1(M)$, $F_2(M)$, ..., $F_n(M)$ are consequently engaged by `M`, using the *Engage-Task* process.

2. if all tasks $F_1(M), F_2(M), \ldots, F_n(M)$ can be achieved, a specific combination of the results of the subtasks, according to the built-in definition of `B`, is performed and the method yields this value.

3. if there is a required task $F_i(M)$ that cannot be achieved, the evaluation of the method fails.

For instance, being `M` a refinement of the `identity?` built-in method, first subtasks `item1` and `item2` will be engaged and then, if those subtasks can be achieved, `M` yields `true` when are the same and `false` otherwise.

The `conditional` built-in method is the only built-in method that engages its required features in a different manner with regard to the previous description. The `conditional` method performs first the subtask `condition` and depending on its result being `true` or `false` either the `result` subtask or the `otherwise` subtask is performed yielding the value inferred by that subtask.

Once a method `M` is evaluated, the inferred value is also automatically cached (see Section 3.5).

```
(>> kind of material-stuff)

Problem Task: kind(<material-stuff>)
Eval: <conditional-1>
⟹
  Task: condition(<conditional-1>)
  Reduce: (>> solid?  made-of)
  ⟹
    Task: made-of(<material-stuff>)
    Value: <thing-1>
    Task: solid?(<thing-1>)
    Value: <true>
  ⟸
  Result: <true>
  Task: result(<conditional-1>)
  Value: <material>
⟸
Result: <material>

<material>
```

Figure 3.12. Inference trace.

An example of Inference

We will illustrate the Noos inference process using a short example. Suppose we define a concept `stuff` with a feature `kind` determining whether a given stuff is `material` or `ideal` using a conditional method as follows:

```
(define Stuff
  ((kind (define (conditional)
           (condition (>> solid?  made-of))
           (result material)
           (otherwise ideal)))))
```

Then, we define a concept `material-stuff` by refinement of `stuff` as follows:

```
(define (stuff material-stuff)
  (made-of (define (thing)
             (solid?  true))))
```

Next, using the following query expression

```
(>> kind of material-stuff)
```

we start the inference of the problem task `kind(material-stuff)`. Since there is a method (that we will call `<conditional-1>`) defined to infer the feature value, it is evaluated. The evaluation of the method performs first the subtask `condition` reducing, in turn, the path reference `(>> solid? made-of)` in the scope of `material-stuff`. Since the result of the subtask `condition` is `true` and the feature value of the `result` subtask of `<conditional-1>` is the constant value `<material>`, the result reduced in the evaluation of `<conditional-1>` is `<material>`. Finally, `<material>` is reduced as the solution for the query expression. Figure 3.12 shows the trace of the inference performed by Noos in this short example.

3.3 Reflection

A *reflective* system is a computational system which is able to reason about aspects of itself. A reflective system has a partial representation of itself that can be inspected and manipulated. The representations the system has of itself are *causally connected* to the system. This means that a change to its self-representation is reflected in the behavior of the system and vice versa. Following [Rademakers, 1988] and [Maes, 1988], a programming language is said to have a *reflective architecture* if it incorporates a framework for implementing reflective systems.

On a formal view, the reflection principles specify the relationship between a theory T and its meta-theory MT. Reflection principles, in turn, are described using three different components: the *upward principles*, the *metalevel inference*, and the *downward principles*. The *upward principles* specify the reification process that encodes some aspects of T into ground facts of MT. That is to say, reification constructs a particular model of T in the language used by MT. The nature of reification and the model constructed is open, i.e. it depends on the purpose for which the reification is made. In logical reflection, the model is about syntactic properties of base-level formulae, so that proof schemas and proof tactics can be the contents of the meta-theory and used to construct strategies for proving base-level formulae [Giunchilia and Traverso, 1990]. Procedural reflection, on the other hand, is based on reifying part of the language semantics for functional languages [Smith, 1985] or for object-oriented languages [Kiczales et al., 1991]. We will use in Noos a knowledge-level model of inference based on tasks and methods. The meta-theory contains knowledge that allows to deduce how to extend the model of the base theory. This deduction process is called *metalevel inference*, and the content of this theory is again specific to the purpose at hand (the meta-theory is indeed no more than a theory). Finally, *downward principles* specify the reflection process that given a new extended model of T has to construct a new theory T' that complies this new model (see Figure 3.13).

There is another thing needed to characterize reflective architectures: *metalevel lift rules*. Metalevel lift rules specify *when* reification and reflection effectively occur, i.e. they specify the control regime. Mainly, there are two classes: (1) *explicit reflection* and (2) *impasse driven reflection* (also called implicit reflection) . In (1) base-level explicitly calls the meta-theory, and in (2) meta-theory is implicitly called when certain situations (impasses) occur at the base-level[4].

[4] Some systems call the meta-level every time a new fact is inferred at the base-level. Al-

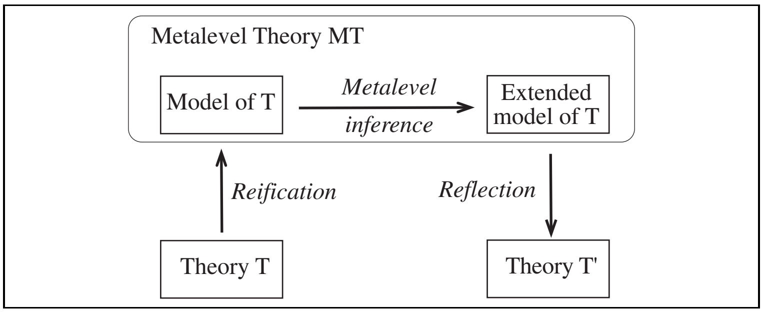


Figure 3.13. The reflection cycle. *Reification* constructs a model of a theory T. *Metalevel inference* infers new facts or takes new decisions that extend (or modify) the model of T using a meta-theory MT. Finally, *reflection* constructs a new theory T' that complies the extended model of T.

As we will see, Noos mainly uses an impasse-driven approach. Nevertheless, Noos provides a collection of metalevel methods that for reasoning using explicit reflection.

In Section 3.1 we have presented the Noos metalevel knowledge as knowledge *about* domain knowledge, problem solving knowledge, and episodic knowledge. Specifically, metalevel knowledge of Noos is modeled as *tasks*, *metalevels*, and *default metalevels*. Tasks, metalevels, and default metalevels are represented in the language as feature terms.

Each feature term is causally connected with one metalevel feature term. A metalevel feature term M is only causally connected with one (base-level) feature term B (called the *referent* of the metalevel).

The features of the referent B have a corresponding feature with the same name on the metalevel M. A feature f of the metalevel M has as feature value the set of methods methods $\{M_i\}$ that are *applicable* to the feature f of the referent B. In other words, since a method is a way to solve a particular task, the set of methods $\{M_i\}$ specifies alternative ways to infer a feature value for f of B. Thus we can conceive of the sets of feature values in a feature f of a metalevel M as a disjunction over the methods that can be used in the feature f of B. A disjunctive expression of methods is used when there is not sufficient knowledge to uniquely determine the method that is guaranteed to solve a specific feature. Noos provides a backtracking mechanism that assures all alternative methods will be reflected down to the referent B and tried if needed. Section 3.4 describes the use of *preferences* as a way to reason and order the evaluation of alternative methods for a given task. Metalevels are described in Section 3.3.1.

A *default metalevel* is a special kind of metalevel. A default metalevel contains a set of methods that can be applied to all the features of a referent and

though it is an extreme case, it can be seen as impasse-driven systems where every new fact causes an impasse.

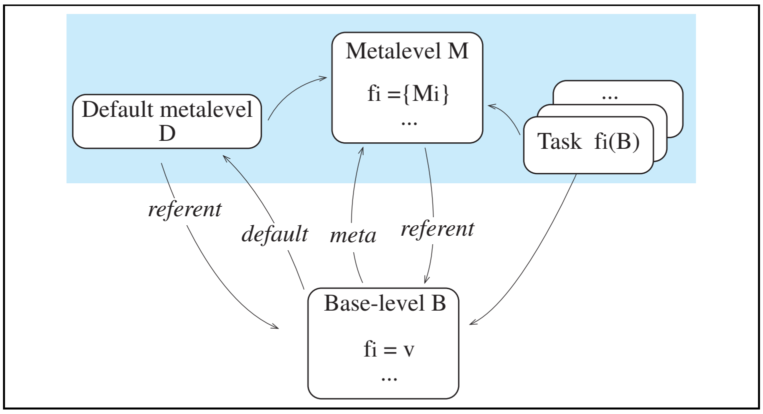


Figure 3.14. Metalevel components of Noos and their causal connections.

are explained in Section 3.3.2. Each feature term can be causally connected with one default metalevel.

Tasks reify the status of the inference in the language. The status of the inference for each feature is reified in the Noos metalevel as a task. A given task T reifies the inference status for a feature f of a base-level B. This task T is causally connected to the base-level B and to the metalevel M of B. Tasks embody episodic knowledge such as the method that has succeeded in achieving that task (the method used to infer the feature value of the feature) and the result of the evaluation of the method (the feature value). Tasks are described in Section 3.3.4.

Noos language provides a set of reflective operations that provides a way to access to the metalevel relations of a given feature term with other feature terms. For instance, reflective operation `meta` applied to a specific feature term B yields the metalevel M with which is causally connected. Reflective operations are described in Section 3.3.5. Some of them are illustrated in Figure 3.14.

Noos is mainly an impasse-driven reflective architecture. The Noos architecture specifies which types of impasses can appear and which kind of metaobject will handle them. For instance, when no method is specified for a given task, a *no-method* impasse occurs and the control of the inference is passed to the corresponding task at the metalevel. Impasses and reflection are described in Section 3.5.

3.3.1 Metalevels

A metalevel in Noos is also a feature term constructed by means of a description. A metalevel description is always a refinement of (predefined) feature term `metalevel`, i.e. the constituent should be `metalevel` or some refinement of it. Metalevels can have a name but usually they are anonymous. An anonymous metalevel is defined as a feature term that has a metalevel causal connection with another (base-level) feature term T (called the *referent* of the metalevel) with the `(meta of` T`)` idiom. That is to say, a metalevel is defined as follows (where usually *metalevel* is `metalevel` itself).

Named Metalevel `(define (`*metalevel NewMetalevel*`)` *body*`)`

Anonymous Metalevel `(define (`*metalevel* `(meta of` T`))` *body*`)`

Since a metalevel feature term is also a feature term, it can have its own metalevel that can be indicated by using two `meta` constructs in its definition, as in `(meta meta of` T`)`. Although this "metalevel tower" can grow in principle as far as needed, in practice only one or two metalevels are used.

Since metalevels are feature terms, feature values of metalevel features can be defined by name references to feature terms, by path references, by anonymous feature terms, or giving a (metalevel) method to compute the feature value. The definition of feature values in metalevels by means of name references allows to define directly a set of alternative methods for a given feature. For instance, the following example:

```
(define (Metalevel (meta of Car))
  (empty-level?  gas-gauge-reading-expl
                 gas-level-in-tank-expl))
```

defines a metalevel that has as referent the `car` feature term. Moreover, it defines the metalevel feature `empty-level?` using two name references to methods `gas-gauge-reading-expl` and `gas-level-in-tank-expl`.

Metalevel feature values can be also defined using a path reference, allowing a metalevel to refer to some methods described in any other feature term. For instance, the nationality of a person can be inferred as follows[5],

```
(define (Metalevel (meta of Person))
  (nationality (>> comes-from of (meta of citizen))))
```

using methods defined for feature `comes-from` in metalevel of `citizen`.

Multiple methods to achieve a subtask

Since methods are also defined as feature terms by means of 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:

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

[5]The `(meta of citizen)` reflective operation is used in path references as a way to refer the metalevel of `citizen` instead of `citizen` (see Section 3.3.5).

There is a simplified syntax that allows to define alternative methods for a feature without the need to explicitly define the metalevel object: the double parenthesis syntax. For instance, the equivalent of the previous metalevel description using double parenthesis is the following:

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

That is to say, a set of methods in a double parenthesis feature is specifying a disjunctive set of applicable methods to that feature. Note that double parenthesis avoids to type the metalevel description but it is, in fact, creating such a metalevel (by refinement of `metalevel`) if it not yet existed.

In fact, we can combine descriptions of methods for some features using the compact syntax and descriptions of methods for other features using a metalevel description. Note that we cannot define methods for a given feature at the baselevel and metalevel at the same time. The Noos interpreter detects this inconsistency and generates an error.

Metalevel Methods

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. Any metalevel method can take into account the information given in the current problem. There are two basic ways in which a method can produce other methods as result: searching for already defined methods or constructing new methods.

In this section we will explain metalevel methods that search for and selects from other methods. Metalevel methods for creating new methods are explained in Chapter 4.7.

In the following example, a metalevel method is defined for the `diagnosis` feature. The metalevel method for `diagnosis` is specified in a way that accesses and obtains different methods for inferring the diagnosis of a car after consulting the `age` of that particular car[6].

```
(define (Metalevel (meta of Car))
  ((diagnosis (define (conditional)
                ((condition (define (bigger-than?)
                              (is-bigger (>> age of (referent)))
                              (than 10))))
                (result (>> usual-malfunctions of (meta of old-cars)))
                (otherwise (>> malfunctions of (meta of new-cars)))))
```

[6]The expression `(referent)` is a reflective operator that obtains 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` (see Section 3.3.5).

Learning methods are examples of methods implemented in Noos by means of metalevel methods. For instance, a CBR metalevel method for `diagnosis` can be defined as a retrieval method that examines previous solved car problems and retrieves 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 (CBR methods and other learning methods are described in Chapter 4).

Multiple Inheritance as Metalevel Inference

We have said on Section 3.2.2 that, using refinement, the features of a constituent description that are not redefined in the new description are "copied" into the new description. In fact, the refinement operation in Noos can be seen as equivalent to single-inheritance with overriding. Moreover, multiple inheritance can be achieved by refinement plus the explicit use of metalevel descriptions.

A simple way to have specialized inheritance is creating a metalevel such that for each feature indicates which methods (defined elsewhere) are to be used. In the following example the methods to be used in `person` are defined to be those defined in metalevels of `citizen` and `homo-sapiens`:

```
(define (metalevel (meta of person))
  (nationality (>> comes-from of (meta of citizen)))
  (children (>> children of (meta of homo-sapiens))))
```

In the example above we show that we can explicitly determine which features "inherit" (reuse) methods of `citizen` and which of `homo-sapiens`, although only two features are shown. Thus, refinements of `person` will reuse methods from `person`, `citizen`, and `homo-sapiens`.

3.3.2 Default metalevels

A *default metalevel* is a special kind of metalevel that applies to *all the features of its referent*. The description of a metalevel is feature-wise: for each feature a method (as a value) or a metalevel method has to be specified. In a default metalevel we can specify a method (or a set of methods) for *any* feature (and *all unspecified features*) of a referent.

A metalevel description is always a refinement of (predefined) feature term `default`, i.e. the constituent should be `default` or some refinement of it. Default metalevels can have a name but usually they are anonymous. An anonymous default metalevel is defined as a feature term that has a default metalevel causal connection with another (base-level) feature term T (called the *referent* of the default metalevel) with the `(default of` T`)` idiom. That is to say, a default metalevel is defined as follows (where usually *default* is `default` itself).

Named Default `(define (`*default NewDefault*`)` *method*+`)`

Anonymous Default `(define (`*default* `(default of` T`))` *method*+`)`

The referent T can be a base-level object or a metalevel (indicated by the construct `(default meta of` T`))`. Whenever a feature f of T is unspecified the method in the default metalevel is installed as *method* for feature f in T. More precisely, it is installed as the value of feature f of the metalevel of T. If the default specifies a set of methods, they are installed as the value of feature f of the metalevel of T. The default metalevel is used only when a feature is unspecified in any of the levels of the metalevel tower. However, only one default can be specified in such a tower. Any other default would be useless since the first one would act on *all* unspecified features.

An example of the use of a default metalevel for analogical reasoning can be found in Section 4.5

3.3.3 Refinement

As we have shown in Section 3.2.2, refinement is used to construct a new feature term reusing another existing feature term. Metalevel knowledge is also reused using refinement. Specifically, a new feature term N defined by refinement of another feature term E as `(define (`E N`)` *body*`)` includes (reuses) all the features defined in E that are not redefined in *body* and, at the metalevel of N, all the features defined in the metalevel of E that are not redefined in the metalevel of N.

For instance, giving the following description of car:

```
(define Car
  (owner (define (person)))
  (price (>> price model)))

(define (Metalevel (meta of Car))
  (empty-level?  gas-gauge-reading-expl
                 gas-level-in-tank-expl))
```

and defining a specific car `Toms-Car` by refinement of `car` as follows:

```
(define (car :id Toms-Car)
  (owner Tom)
  (model Ibiza))
```

and defining also a metalevel for `Toms-Car`

```
(define (Metalevel (meta of Toms-Car))
  (fault-symptoms model-based-method
                  empirical-method))
```

then, both features `price` and `empty-level?` are included in `Toms-Car` from `car` and from metalevel of car respectively. Feature `price` is included in `Toms-Car`. Feature `empty-level?` is included in the metalevel of `Toms-Car`.

Default metalevels are also reused by refinement in a similar way to metalevel descriptions of features.

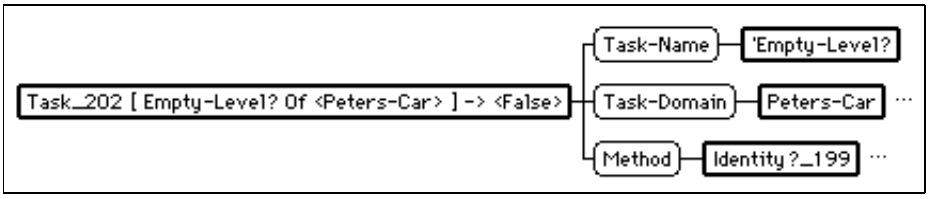


Figure 3.15. Task feature term reifying the inference of feature `empty-level?` of `Peters-Car`. Note that the printname of the task contains the referent.

3.3.4 Tasks

A `task` is a feature term that reifies the current state of the inference for a given feature. Tasks are built automatically by Noos. A task embodies three features: `task-name`, `task-domain`, and `method`. Tasks cannot be defined by descriptions.

Feature `task-name` keeps the feature name of the feature that it reifies. Feature names are represented in Noos as symbols (e.g. `'Empty-level?`). Feature `task-domain` keeps the feature term in which appears the feature that reifies. Feature `method` keeps the method that has succeeded in achieving that task.

For instance, given a feature term `D` with one feature `f` built from the following description

```
(define D
  ((f M)))
```

the feature value of feature `task-name` of the task built by Noos is `f`, feature value of feature `task-domain` is `D`, and the feature value of feature `method` is `M`, assuming that `M` has succeeded in achieving the feature value for `f`. Figure 3.15 shows the task built by Noos for the feature `empty-level?` of `Peters-car` feature term.

Tasks allow Noos programs to inspect its own inference status. The task of a given feature can be introspected using reflective operations `task` and `current-task` (see Section 3.3.5). Since tasks are feature terms, their features can be inspected using path references.

3.3.5 Reflective operations

Reflective operations allows to know the metalevel relations of a given feature term with other feature terms.

Meta

Given a reference to a feature term *term*, the `meta` reflective operation

(`meta` *term*)

refers to the metalevel of *term*. The reference to a feature term *term*, that is optional, can be a name reference, a path reference, or a reflective operation. For instance, the following expression

```
(meta Peter)
```

is a reference to the metalevel of `Peter`. Another example is using a path reference as follows

```
(meta (>> wife of Peter))
```

that is a reference to the metalevel of the `wife` of `Peter`. Assuming that, for instance, the wife of `Peter` is `Mary` this metalevel operation defines a reference to the metalevel of `Mary`.

The metalevel of any feature term can be always obtained. When a metalevel is requested and it was not previously defined, it is created.

When the *term* is not specified it is determined by the rules of scope and refinement (see Section 3.2.3).

Default

Given a reference to a feature term *term*, the `default` reflective operation

(`default` *term*)

allows to refer to the metalevel default of *term*. The reference to a feature term *term* can be a name reference, a path reference, or a reflective operation. For instance, the following expression

```
(default Peter)
```

is a reference to the default metalevel of `Peter`.

Default metalevels are optional. This means that not all feature terms have a default metalevel. When the `default` metalevel operation is performed over a feature term without a default metalevel, the `default` metalevel operation fails (see Section 3.5 for the definition of failure).

Task

Given a feature name *f* and given a reference to a feature term *term*, the `task` reflective operation

(`task` *f* `of` *term*)

allows to refer to the task feature term that reifies the inference status for a feature *f* of that feature term. For instance, the following expression

```
(task diagnosis of Peters-car)
```

is a reference to the task feature term that reifies the inference status for a feature `diagnosis` of `Peters-car`.

Current-task

Reflective operation `current-task` allows to refer to the task in which a method is involved.

(`current-task` *method*)

Since the task is a term with the `task-domain` feature holding the feature term in which the method is inferring a feature value, `current-task` is used in Noos as a way to access directly to other features of the feature term. This alternative avoids the use of parameters in the method.

For instance, we define the `adult?` method as a method that directly accesses to the feature value of the `age` feature, and returns `true` if the age is higher to 17 and `false` otherwise.

```
(define (higher-than adult?)
  (is-higher (>> age task-domain of (current-task)))
  (than 17))
```

Then, we define a specific person using the `adult?` method for inferring the feature value of feature `can-vote?` as follows:

```
(define Carol
  (age 22)
  ((can-vote?  adult?)))
```

Finally, the following query expression:

```
(>> can-vote?  of Carol)
```

yields `true` since the age of `Carol` is 22 years old.

Referent

The `referent` reflective operation is used with the following syntax:

(`referent` *term*)

where *term* follows also the previous defined alternatives.

The `referent` reflective operation performs different references depending on the feature term from which it is applied. The referent of a metalevel feature term M refers to the feature term T of which M is the metalevel. For instance, the `referent` of the `metalevel` of `Peter` is expressed as follows:

```
(referent (meta Peter))
```

Clearly, the referent of the metalevel of `Peter` is `Peter`.

The `referent` of a default metalevel feature term D refers to the feature term T of which D is the default metalevel. For instance, `Peter` is the referent of the default metalevel of itself an is expressed as follows:

```
(referent (default Peter))
```

The `referent` of a task T that reifies the inference status for a feature f of a feature term B, refers to the result of the evaluation of the method used to infer the feature value of f. For instance, assuming that the feature value for the `empty-Level?` feature of `Peters-Car` is the feature term `false`, the following expression

```
(referent (task empty-level?  of Peters-car))
```

is a reference to the feature term `false`.

Combining reflective operations with path references

Since reflective operations are references to feature terms, reflective operations can be used as references in path references. Specifically, path references and reflective operations can be combined as follows:

comb	::=	`(>>` *feature-name** `of` comb`)`
	|	`(meta` comb`)`
	|	`(default` comb`)`
	|	`(task` *feature-name* `of` comb`)`
	|	`(current-task` comb`)`
	|	`(referent` comb`)`
	|	*name*

For instance, the following path reference

```
(>> diagnosis of (meta Peters-car))
```

is a referent to the set of alternative methods that are applicable to the feature `diagnosis` of `Peters-car` that are embodied in the metalevel of `Peters-car`.

3.3.6 Reification

Reification is the process by which an expression is converted into an object (a value) of a particular language. In Noos there are two kinds of reification: reification of path references, and reification of method evaluation.

Reification of Path References

In Noos, path references can be reified into the language as *query-methods*. Noos provides four kinds of built-in query-methods: `Infer-value`, `Exists-value`, `Known-value`, and `All-values`. `Infer-value` is a method that reifies the inference process involved in the reduction of a path reference. The rest three query-methods provides a set of basic metalevel inference capabilities about feature values. Query-methods allow to define path references using method descriptions.

Infer-value

The `Infer-value` method has two required features: `feature` and `domain` (all the query-methods have these two required features). `Infer-value` reifies path reduction inference. Its evaluation engages first the `feature` subtask (for obtaining a feature name `F`), next engages the `domain` subtask (for obtaining a feature term `D`), and finally engages the `F(D)` task (see Section 3.2.5) and the evaluation of *Infer-value* method yields the value inferred by task `F(D)`. Feature names are represented as quoted symbols.

For instance, the following path reference defined in the feature `father` of `Person`:

```
(define Person
  (father (>> husband mother)))
```

can be reified as an `infer-value` method in the following way:

```
(define Person
  ((father (define (infer-value)
             (feature 'husband)
             (domain (>> mother))))))
```

Finally, the remaining path reference `(>> mother)` can also be reified as a method. In this case, the expression above is equivalent to the one defined as a composition of two `infer-value` methods in the following way:

```
(define Person
  ((father (define (infer-value)
             (feature 'husband)
             ((domain (define (infer-value)
                        (feature 'mother)
                        (domain (>>)))))))))
```

We have seen that a path reference can also be defined as a method. This process can be performed directly using the `reify` construct. The `reify` construct takes a path reference and builds an `infer-method` that reifies the path reference.

For instance, the following path reference

```
(>> father mother of Peter)
```

can be reified using the `reify` construct as follows

```
(reify (>> father mother of Peter))
```

yielding an `infer-value` method equivalent to the following:

```
(define (infer-value)
  (feature 'father)
  ((domain (define (infer-value)
             (feature 'mother)
             (domain Peter)))))
```

The `reify` construct is handy when we require to have multiple alternative references in a feature. In the following example, a person may be located by a phone number, but there are several phone numbers where she could be found. An easy way to model this situation is to have a disjunctive set of path references to different phone numbers as follows:

```
(define (person professional)
  ((phone-number (reify (>> phone-number spouse))
                 (reify (>> phone-number home))
                 (reify (>> phone-number works-in)))))
```

As this example shows, the `phone-number` feature has at the metalevel three path references reified as methods. Since they are methods, we can interpret them as three alternative ways to find out the phone number where a professional can be located.

The three other query-methods provide a set of basic metalevel inference capabilities about feature value inference. These query-methods allows to reason about the value of any feature `F` of a term `D` without neither modifying the value nor engaging the inference in the task `F(D)`.

Before to explain these other query-methods, we will extend the syntax of path references in order to provide to all the four query-methods a syntax based on path references as follows:

path-reference	::=	(>> *feature-name** [*of name*])
	|	(?>> *feature-name** [*of name*])
	|	(!>> *feature-name** [*of name*])
	|	(*>> *feature-name** [*of name*])

where a path reference starting with the token `>>` corresponds to a `infer-value` method as we have seen. Moreover, a path reference starting with the token `?>>` corresponds to an `exists-value` method; a path reference starting with the token `!>>` corresponds to a `known-value` method; and a path reference starting with the token `*>>` corresponds to an `all-values` method.

Exists-value

`Exists-value` is a query-method that determines if the feature value for a given feature `F` of a feature term `D` can be inferred—or in other words, if the task `F(D)` can be achieved. The evaluation of *Exists-value* method yields `true` if there is at least one method that succeeds in achieving task `F(D)`, and `false` otherwise. Note that *Exists-value* method does not yield the value that can be inferred.

For instance, we can define a `bird` with a feature `can-fly?` as follows:

```
(define Bird
  ((can-fly?  (define (conditional)
               (condition (?>> exceptional-bird?))
               ((result (define (not)
                          (item (>> exceptional-bird?)))))
               (otherwise true)))))
```

specifying that a bird can fly excepting when it is defined as an exceptional bird. Note that the feature value of feature `can-fly?` will yield always a feature value without forcing to define the feature `exceptional-bird?` for all refinements of `bird`. For instance, we can define a specific bird's species as follows:

```
(define (bird sparrow))
```

where the `exceptional-bird?` feature is still undefined.

Then, performing the query expression `(>> can-fly? of sparrow)`, value yielded is `true` because path reference `(?>> exceptional-bird?)` will yield `false` (since there is no way to infer a value for `exceptional-bird?` feature).

On the other hand, defining penguins as follows:

```
(define (bird penguin)
  (exceptional-bird?  true))
```

and performing the query expression `(>> can-fly? of penguin)`, we will obtain `false` as answer because path reference `(?>> exceptional-bird?)` will yield `true` (since there is a value for `exceptional-bird?` feature) and the negation of value for `exceptional-bird?` feature is `false`.

Known-value

Known-value is a query-method that determines if the feature value for a given feature `F` of a feature term `D` is already *known*. Since inference in Noos is lazy, a feature value is known only if the feature value is a constant or it has been previously inferred—or in other words, task `F(D)` has been previously achieved. The required features of a `known-value` method are also the `feature` and `domain` features. The evaluation of *Known-value* method yields `true` whenever task `F(D)` has been previously achieved and `false` otherwise. Note that *Known-value* method neither does yield the value of `F(D)`.

A `known-value` method can be used to check the inference status in a given task. For instance, suppose that in a specific step of the inference we have two alternative methods m_1, m_2 to achieve a subtask. We know that one of them (for instance m_1) requires a knowledge source that is complex to acquire. Then, we can check first whether this source has been inferred previously using the `known-value` method and choose m_1 only when this source is already available and choose m_2 otherwise.

All-values

`All-values` is a query-method that, for a given task `F(D)`, determines the set of all feature values that can be inferred for `F(D)`—in other words, the set containing the values that result from all the methods that may succeed in achieving that task.

For instance, defining `Carol` by refinement of `professional` as follows:

```
(define (professional Carol)
  (spouse (define (person)
           (phone-number 3344)))
  (works-in (define (company)
             (phone-number 8766))))
```

and performing the query expression `(*>> phone-number of Carol)` we will yield the set of phone numbers `<set of 3344 8766>`. Note that the `phone-number` feature of a `professional` was defined in page 66 with three path references. The reference to the phone number of Carol's home did not succeed, so only two phone numbers are inferred by `all-values`.

Reification of method evaluation

The process of method evaluation can be reified also in the language. Method evaluation is reified by means of the `noos-eval` method. The `noos-eval` method has one required feature called `methods`. The evaluation of a `noos-eval` method engages first the subtask `methods` for obtaining a method (or a set of methods) to be evaluated, and then engages the evaluation of that method. For instance, the evaluation of the following method defined for inferring the feature value of feature `can-vote?` of `Person`:

```
(define Person
  ((can-vote?  (define (higher-than)
                (is-higher (>> age))
                (than 17)))))
```

can be also reified using a `noos-eval` method as follows:

```
(define Person
  ((can-vote?  (define (noos-eval)
                (methods (define (higher-than)
                           (is-higher (>> age))
                           (than 17)))))))
```

Reification can be also performed to query expressions. 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 can be reified into a `noos-eval` method with a feature `methods` whose value is the query-method corresponding to the reification of the original query expression, as follows:

```
(define (noos-eval)
  (methods (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)
  (methods (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`, `Exists-eval`, `Known-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.

The `Exists-eval` method determines if it is possible to evaluate successfully a method; `Exists-eval` yields `true` if it is possible and `false` otherwise.

The evaluation of a `Known-eval` method determines if a method `M` has been successfully previously evaluated; `Known-eval` yields `true` when `M` has been successfully previously evaluated and `false` otherwise.

Finally, the evaluation of an `All-eval` method giving a specific method `M` yields the set of results of all the successful evaluations of method `M`.

3.3.7 Reinstantiation

Each time a method `M` for solving a task `F(D)` is reflected down to `D` from the metalevel term of `D`, `M` is reinstantiated and bound in the context of `D`. The reinstantiation mechanism can be understood as refinement: a new method `M'` is built by refinement of method `M`, and relative path references are bound in the context of `D`.

The calculus of the scope of a relative path reference is performed taking into account three cases:

- When method `M` was defined in the context of `D` or in the context of its metalevel, the new method `M'` is a refinement of `M` where relative path references have not to be changed;
- When method `M` was defined alone (as a root description), the new method `M'` is a refinement of `M` where relative path references to `M` are bound to `M'`; and
- When method `M` is a closed method defined in the context of another term `D'`, the new method `M'` is a refinement of `M` where relative path references are bound following refinement scope rules (see Section 3.2.3).

For instance, suppose that we define the `phone-number` of a `professional`, as defined in Section 3.3.6, as follows:

```
(define Professional
  ((phone-number (reify (>> phone-number spouse))
                 (reify (>> phone-number home))
                 (reify (>> phone-number works-in)))))
```

Next, we define `Ann` as a person that works in a particular company, and with a feature `phone-number` using the methods described in `professional` as follows:

```
(define (Person Ann)
  (works-in (define (company)
              (phone-number 2627))))

(define (metalevel (Meta of Ann))
  (phone-number (>> phone-number of (meta Professional))))
```

Methods defined for feature `phone-number` in `professional` will be reflected down in the context of `Ann` using refinement and bound to `Ann`. Thus, posing the following query expression to Noos:

```
(>> phone-number of Ann)
```

one of the three methods succeeds yielding `2627` as result.

The automatic reinstantiation mechanism of Noos provides a powerful mechanism for integrating learning methods (see Chapter 4).

3.4 Preferences

Preferences in Noos are a declarative mechanism for decision making about sets of alternatives present in domain knowledge and problem solving knowledge. The main usages of preferences in Noos are:

- As a declarative control construct for search and backtracking—by determining the order in which a metalevel task chooses a method for a task from a set of alternative methods.
- As a symbolic representation of relevance (or "similitude") in comparing a given current problem with problems previously solved by the system (also called precedents).

As we have shown in Section 3.3.1, we can define a set of alternative methods to solve a given task. Preferences provide a declarative mechanism for ranking a set of alternative methods. Specifically, preference knowledge can be used (1) for determining a fixed order of execution of methods in a given task, or (2) for dynamically calculating an execution ordering of methods according of the knowledge available for each problem. Notice that in our approach preferences are local to some task

Preferences are also used as a symbolic representation of relevance in retrieval and selection of precedents in case-based reasoning. For instance, preference knowledge can be used to model criteria for ranking some precedent cases over other precedent cases for a task in a specific situation.

Preferences over sets are modeled by partially ordered sets (also called *posets*). A partially ordered set is a pair $\langle S, \prec \rangle$ composed by a set of elements S and a partial order relation $\prec$ defined on S. When $a \prec b$ we say that a is *preferred* to b. Preferences are described formally in Section 5.10.

```
(define (personal-computer PC-blue)
  (freq-MHz 150)
  (disk-capacity-Gb 1)
  (monitor color-14)
  (price-$ 4000))

(define (personal-computer PC-red)
  (freq-MHz 133)
  (disk-capacity-Gb 2)
  (monitor color-14)
  (price-$ 4000))

(define (personal-computer PC-white)
  (freq-MHz 200)
  (disk-capacity-Gb 2)
  (monitor color-15)
  (price-$ 6000))
```

Figure 3.16. Specification of characteristics of three personal computers `PC-blue`, `PC-red`, and `PC-white`.

Preference methods

Preferences in Noos are built by means of *preference methods*. A preference method takes a set of source elements and an ordering criterion and builds a partially ordered set (for the sake of brevity, the poset that is the result of such a method it will be simply called a *preference*). Since preference methods are Noos methods, they can be used as any other method. Different preference methods correspond to different ordering criteria.

There are several built-in preference methods in Noos. A built-in preference method is `decreasing-preference`. `Decreasing-preference` takes a set of elements in feature `set` and the identifier of a numeric feature in feature `feature-name` and builds a preference where the preferred elements are those with a lesser value in the specified feature. Another built-in preference method is `increasing-preference` that builds a preference for higher values with respect to lesser values in a similar way to `decreasing-preference`.

For instance, given the computer descriptions in Figure 3.16 three different preference criteria based on numerical values of feature values can be built. A first preference method `cheaper-pref` is built by a refinement of the built-in method `decreasing-preference`. Preference method `cheaper-pref` builds an ordering based on prices of computers, for the set of computers given in the feature `set`, resulting in a partially ordered set where PCs are preferred from cheaper to more expensive.

```
(define (decreasing-preference cheaper-pref)
  (set PC-red PC-blue PC-white)
  (feature-name 'price-$))
```

The preference obtained using this `cheaper-pref` method is shown in Figure 3.17(a).

Another preference method `disk-pref` is built by a refinement of the built-in method `increasing-preference` that constructs a preference, in a similar way,

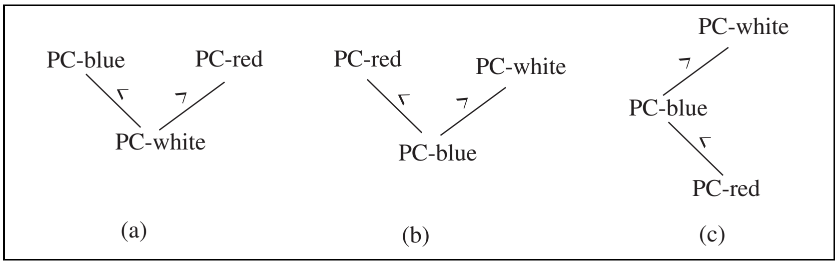


Figure 3.17. Graphical representation of three different preferences over computers. (a) corresponds to `cheaper-pref`, (b) is built using `disk-pref`, and (c) is obtained with `faster-pref`.

using the numeric feature `disk-capacity-Gb`.

```
(define (increasing-preference disk-pref)
  (set PC-red PC-blue PC-white)
  (feature-name 'disk-capacity-Gb))
```

Using this preference we are establishing an order where the preferred computers are those with a higher disk capacity (see Figure 3.17(b) for a graph representation of the preference).

Finally, we can define the `faster-pref` preference method (also based on method `increasing-preference`) for feature `freq-MHz` providing a preference from faster to slower computers using CPU clock rate as estimator (see Figure 3.17(c) for a graph representation of the preference).

```
(define (increasing-preference faster-pref)
  (set PC-red PC-blue PC-white)
  (feature-name 'freq-MHz))
```

Noos provides other numerical and non-numerical preference methods. Examples of non-numerical preference methods are `equal-value-preference` and `subsumption-preference` (the complete list of preference methods is explained in Appendix D). For instance, using the preference method `equal-value-preference` in the computers example we can establish a preference over a specific kind of monitor (a preference for `color-14` inch monitor or for `color-15` inch monitor).

Preference combination

There are several ways to combine different preference criteria—or, in other words, building new preferences from existing preferences. The Noos operations dealing with preference combination are methods that create new partially ordered sets from (a combination of) partially ordered sets—created either by preference methods or by other preference combination methods. Examples of preference combinations are operations such as *inversion*, *preference union*, and *preference intersection*. In this section we will only describe some of them, but all preference combination operations are explained in Appendix D.

Inversion takes a preference P and builds a new preference P' where all the relations $a \prec b$ defined in P are inverted in P' ($b \prec a$). For instance, applying the inversion of an order obtained by the `increasing-preference` is equivalent to directly applying the `decreasing-preference`.

Preference union takes two preferences and constructs a new preference performing a union of the sets and a transitive closure of the union of order relations. The transitive union method is called `t-union` in Noos. For instance, we express a criterion of preferring either cheaper computers or computers with high disk capacity, combining with `t-union` the preference based on prices of computers `cheaper-pref` and the preference based on disk capacities `disk-pref` as follows:

```
(define (t-union cheaper&disk-pref)
  ((poset1 (define (cheaper-pref))))
  ((poset2 (define (disk-pref)))))
```

The result obtained with this preference criterion is shown in Figure 3.18(a).

Another possibility is to combine the preference based on prices `cheaper-pref` with the preference based on CPU clock rate `faster-pref`, expressing a preference criterion of preferring either cheaper or faster computers:

```
(define (t-union cheaper&faster-pref)
  ((poset1 (define (cheaper-pref))))
  ((poset2 (define (faster-pref)))))
```

The result obtained with this preference criterion is shown in Figure 3.18(b).

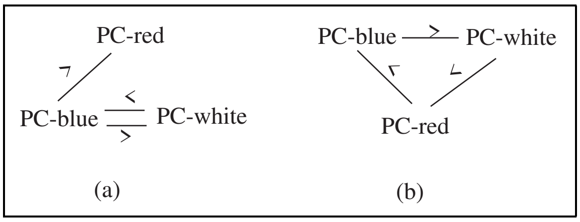


Figure 3.18. Combining preferences.

Taking the first combination `cheaper&disk-pref` (Figure 3.18(a)) we obtain that the most preferred computer is `PC-red` and that `PC-blue` and `PC-white` are equally preferred—since `PC-blue` < `PC-white` and `PC-white` < `PC-blue`. The problem given with the second combination `cheaper&faster-pref` (Figure 3.18(b)) is that CPU clock rate and prices introduce inverse preferences causing a cycle where all the three computers are equally preferred. In other words, this situation causes an indeterminism of preferences. In order to avoid this problem we have to use higher order preferences as will be explained in next section.

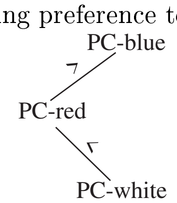
Higher order preferences

As we have shown, preference combination operations are modeled as methods. This uniform representation of Noos allows to model higher order preferences also as preference methods that build preferences from preferences over preferences.

A higher order preference operation is the preference combination method *hierarchical union* (called `h-union` in Noos). This combination preference is used when we have the knowledge that a preference is more important than (is preferred to) a second preference. This preference method—given a more preferred poset in feature `higher-poset` and a less preferred poset in feature `lower-poset`—constructs a preference order preserving the order fixed in `higher-poset` and adding from `lower-poset` the order relations that are not in conflict with `higher-poset`. For instance, considering the price as a preference more important than the CPU clock rate we can define the following hierarchical combination of preferences:

```
(define (h-union cheaper-faster-pref)
  ((higher-poset (define (cheaper-pref))))
  ((lower-poset (define (faster-pref)))))
```

obtaining as a result the following preference total order:



where `faster-pref` preference is used to discriminate between computers with the same cost (`cheaper-pref`). In this last example the most preferred computer will be `PC-blue`.

Notice that the uniform representation of Noos allows to have preferences over higher order preferences, and potentially no limit of preferences over preferences could be built.

3.5 Inference in Noos

In Section 3.2.5 we have described the basic inference process of Noos. Now, we will complete the description of the Noos inference process incorporating metalevel reasoning and preference-based decision taking.

As we have seen in Section 3.2.5, inference in Noos is on demand and starts when the user poses a query to the system by means of a *query expression* that engages a problem task `F(D)`. We said then that, when neither a path reference nor a method is defined for a task `F(D)`, an *impasse* occurs and the control of the inference is passed to the metalevel.

Solving an impasse for a task `F(D)` involves three processes: (i) determining a set of methods $\{M_i\}_{F(D)}$ applicable to task `F(D)`, that can be partially ordered with preferences, (ii) selecting a method from $\{M_i\}_{F(D)}$, according to

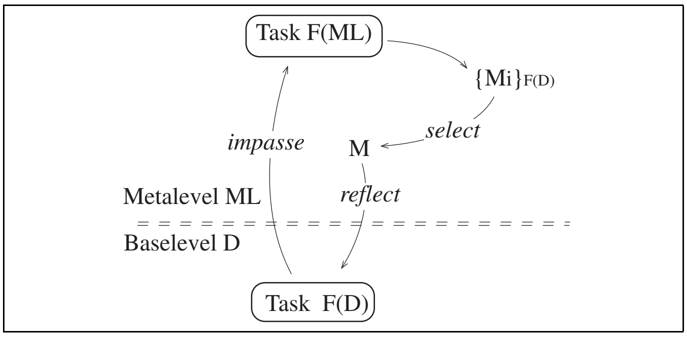


Figure 3.19. Solving and *Impasse* for a task `F(D)` in the metalevel `ML` of `D`.

the preferences, and (iii) reflecting down the selected method to task `F(D)` (see Figure 3.19).

Reflection ensures that:

1. when a method `M` is reflected down to a task `F(D)`, `M` is reinstantiated and correctly bound in the context of `F(D)`;

2. if method `M` fails in solving a task `F(D)`, backtracking is engaged and one of the remaining non-failed methods in $\{M_i\}_{F(D)}$ will be reflected down;

3. if there is a preference on the set of alternative methods $\{M_i\}_{F(D)}$, any method reflected down is maximally preferred among the non-failed methods in $\{M_i\}_{F(D)}$.

Moreover, since a method `M` for `F(D)` can have subtasks, and each subtask may have several alternative methods to solve it, metalevel inference ensures that the possible combinations of methods for each subtask are tried, following the local preference orderings for each subtask, until a solution is found. Furthermore, metalevel inference ensures that all these combinations are tried before declaring that a method `M` fails.

3.5.1 Metalevel methods

When an impasse occurs in solving a task `F(D)`, the first process involved is to determine a set of methods $\{M_i\}_{F(D)}$ applicable to task `F(D)`. Each feature `F` in `D` has a corresponding feature with the same name `F` on the metalevel `ML` of `D` (see Section 3.3). The value of this feature `F` at the metalevel has the set of methods $\{M_i\}_{F(D)}$ applicable to task `F(D)`. This set of methods can be defined by name references, by a path reference, or by giving a (metalevel) method for inferring the methods. When an impasse occurs in solving a task `F(D)` and a metalevel method `MM` is defined for determining the set of methods applicable to task `F(D)`, a metalevel task `F(ML)` is engaged for evaluating the metalevel method `MM`.

Notice that, since a metalevel task `F(ML)` is also a task, a new impasse may occur in solving task `F(ML)`. When an impasse occurs at the metalevel, the control of the inference is passed to the metalevel of the metalevel.

3.5.2 Caching

Once the task `F(D)` of inferring the value for a feature `F` of a term `D` is achieved by a method `M`, the inferred value and method `M` are automatically stored in the task term that reifies task `F(D)` in the language (see Section 3.3.4). This caching mechanism allows Noos to quickly answer future demands of already inferred feature values.

In order to maintain the consistency of values inferred in tasks and used in different methods, only the method `M'` that first engages a task `F(D)` can perform backtracking on this task. We call such method the *owner* of task `F(D)` and we also say that `F(D)` is engaged by `M'`.

When the *owner* of a task forces backtracking and no other possible value can be inferred for this task, the task is disengaged. After that, any other method can engage that task.

3.5.3 Backtracking

Three kinds of backtracking are engaged by Noos inference involving backtracking on tasks and on methods:

1. Backtracking on tasks is engaged when a method `M` fails in solving a task `F(D)`. Then, the control of the inference is passed to the metalevel and one of the remaining non-failed methods in $\{M_i\}_{F(D)}$ is selected. The selected method is reflected down and the inference is resumed in task `F(D)`. When no more methods can be selected, we say that the task `F(D)` cannot be achieved.

2. Backtracking on subtasks of methods is engaged when a subtask of a method cannot be achieved. Let us assume that during the evaluation of a method `M` its subtasks $\mathtt{F}_1\mathtt{(M)}$, ..., $\mathtt{F}_i\mathtt{(M)}$ have been achieved and that task $\mathtt{F}_{i+1}\mathtt{(M)}$ cannot be achieved from the results of the previous achieved subtasks. Then, backtracking is engaged for inferring another value for subtask $\mathtt{F}_i\mathtt{(M)}$. The value of subtask $\mathtt{F}_i\mathtt{(M)}$ can be constant, inferred by a path reference, or inferred by a closed method. When the value of subtask $\mathtt{F}_i\mathtt{(M)}$ is constant, backtracking is recursively engaged for inferring another value for subtask $\mathtt{F}_{i-1}\mathtt{(M)}$. When subtask $\mathtt{F}_i\mathtt{(M)}$ has been achieved by a closed method $\mathtt{M}_i$—or a path reference—backtracking on $\mathtt{F}_i\mathtt{(M)}$ will involve recursively backtracking on method $\mathtt{M}_i$ (see third kind of backtracking). Backtracking on method $\mathtt{M}_i$ may either succeed or fail: when it succeeds, the evaluation of method `M` is resumed at subtask $\mathtt{F}_{i+1}\mathtt{(M)}$; when it fails,

backtracking on task $\texttt{F}_i\texttt{(M)}$ is engaged (see first kind of backtracking). Finally, when backtracking on the first subtask $\texttt{F}_1\texttt{(M)}$ of a method M for solving a task F(D) fails, we say that the method M fails in achieving F(D).

3. Backtracking on a method M is engaged when a new value for the task F(D) that M solves is required. Let us assume that backtracking is engaged in a method M having subtasks $\texttt{F}_1\texttt{(M)}$, ..., $\texttt{F}_m\texttt{(M)}$ are achieved. Then, backtracking is engaged for inferring another value for the last subtask $\texttt{F}_m\texttt{(M)}$ and backtracking is resumed as described in previous kind. Thus, backtracking on methods also involves backtracking on subtasks of a method.

3.5.4 The Noos inference engine

In Section 3.2.5 the Noos inference engine was described using three basic processes: the *Engage-Task* process, involving the inference of feature values; the *Reduce-Path-Reference* process, involving the reduction of a path reference; and the *Noos-Eval* process, involving the evaluation of a method.

Moreover, we have seen in Section 3.3.6 that path references can be reified as query-methods. Using this property, the Noos inference engine reifies path references as methods. Thus, the *Reduce-Path-Reference* process is considered as a specific kind of *Noos-Eval* process and we will dispense of it in what follows.

We will now complete the description of the Noos inference engine by including the inference process involved in solving an impasse and in backtracking. Impasses engage a metalevel process that we will call *No-Method-Impasse*. Backtracking is engaged by three processes: *Next-Value*, involving backtracking of inference of feature values; *Failed-Method-Impasse*, involving backtracking of metalevel inference; and *Eval-Next*, involving backtracking of the evaluation of methods.

In summary, the Noos inference engine can be described using six basic processes: *Engage-Task* and *Next-Value* processes, involving the inference of feature values; *No-Method-Impasse* and *Failed-Method-Impasse* processes, involving metalevel reasoning; and *Noos-Eval* and *Eval-Next* processes, involving the evaluation of a method.

Engage-Task

The *Engage-Task* process is engaged by a problem task or by solving the subtasks of a method in the evaluation of that method. The goal of the *Engage-Task* process is to infer the value of a feature. *Engage-Task* process involves three different actions according to the different specifications of a feature value as follows:

Given a task F(D) engaged for determining the value of a feature F of a feature term D,

1. When there is a constant value or a cached feature value C (see Section 3.5.2 for an explanation of cached values) for feature F of D, the task F(D) is directly achieved yielding C.

2. When there is a closed method [7] `M` defined for feature `F` of `D`, the *Noos-Eval* process is engaged for method `M`. The task `F(D)` is achieved if the evaluation of method `M` succeeds, yielding the value inferred in that evaluation. Otherwise *Engage-Task* fails.

3. When neither a value nor a method is defined for feature `F` of `D`, an *impasse* occurs and the *No-Method-Impasse* process is engaged. If *No-Method-Impasse* reflects down a method `M`, then the *Noos-Eval* process is engaged for method `M`. Otherwise *Engage-Task* fails.

   (a) If the evaluation of method `M` succeeds, the task `F(D)` is achieved yielding the value inferred in that evaluation.

   (b) Otherwise another *impasse* occurs and the *Failed-Method-Impasse* process is engaged for reflecting down another method `M'`. Then, *Noos-Eval* and *Failed-Method-Impasse* processes are iterated until the evaluation of a method succeeds or there are no more methods to reflect down (*Failed-Method-Impasse* fails). If the evaluation of a method succeeds, task `F(D)` is achieved yielding the value inferred in that evaluation. Otherwise *Engage-Task* fails.

No-Method-Impasse

The *No-Method-Impasse* process is engaged when neither a value nor a method is defined for a task `F(D)`. Then, the control of the inference is passed to the metalevel. The goal of the *No-Method-Impasse* process is to select a method, from an alternative set of methods $\{M_i\}_{F(D)}$, for solving task `F(D)`. Being `ML` the metalevel of `D`, the set of methods $\{M_i\}_{F(D)}$ can be defined directly in the metalevel task `F(ML)` or can be inferred by engaging a metalevel method defined in `F(ML)`. If *No-Method-Impasse* fails means that no method M_i, inferrable at the metalevel `ML`, exists that can solve `F(D)` given current engagements.

No-Method-Impasse involves four different actions:

Given an impasse generated in solving a task `F(D)`, and being `ML` the metalevel term of `D`,

1. First, the *Engage-Task* process is engaged for solving the metalevel task `F(ML)`.

2. When metalevel task `F(ML)` is achieved it yields a set of partially ordered alternative methods $\{M_i\}_{F(D)}$ for solving task `F(D)`. Otherwise `F(ML)` could not be achieved and the *No-Method-Impasse* process fails.

3. Then, one of the methods `M` maximal in $\{M_i\}_{F(D)}$, according to the preference, is selected.

[7] Note that path references are reified as methods. Thus, path references are also defined as a closed method for feature `F` of `D`.

4. Finally, the selected method `M` is reflected down to task `F(D)`. The reflection process reinstantiates method `M` in the context of task `F(D)` (see Section 3.3.7).

After this impasse is resolved, the inference is resumed in the *engage-task* process engaged for solving task `F(D)`.

Noos-Eval

The *Noos-Eval* process for a method `M` first engages a task for each required feature of `M` and then performs a specific combination of the results of the subtasks according to the built-in method it is a refinement of.

Specifically, given a method `M` that is a refinement of a built-in method `B` with required features F_1, F_2, ..., F_n,

1. tasks $F_1(M)$, $F_2(M)$, ..., $F_n(M)$ are consequently engaged by `M`, using the *Engage-Task* process. When a task $F_i(M)$ ($i > 1$) cannot be achieved (assuming $F_1(M) \cdots F_{i-1}(M)$ have been achieved) backtracking is engaged in task $F_{i-1}(M)$ using the *Next-Value* process. Backtracking on task $F_{i-1}(M)$ can succeed or can fail: when it succeeds, the evaluation of method `M` is resumed at subtask $F_i(M)$; when it fails, backtracking on task $F_{i-2}(M)$ is recursively engaged. Finally, when backtracking on the first subtask $F_1(M)$ fails, the evaluation of the method fails.

2. When all tasks $F_1(M)$, $F_2(M)$, ..., $F_n(M)$ are achieved, a specific combination of the results of the subtasks, according to the built-in definition of `B`, is performed and the method yields this value.

Next-Value

The *Next-Value* process is engaged when backtracking is forced in a task `F(D)` because the value currently inferred for `F(D)` is not adequated for another task. The goal of the *Next-Value* process is to force backtracking and infer another possible value of task `F(D)`. If *Next-Value* fails, then no value can be inferred for task `F(D)`, given current engagements.

Given a term `T` requesting a task `F(D)`, *Next-Value* determines another possible value for feature `F` of a feature term `D` involving two different actions:

1. When there is a constant feature value or task `F(D)` has an owner different to term `T`, *Next-Value* fails.

2. When there is a closed method `M` defined for task `F(D)`, the *Eval-Next* process is engaged for method `M`.

   Task `F(D)` is achieved if the evaluation of method `M` succeeds, yielding the value inferred in that evaluation.

   If the evaluation of method `M` fails, an *impasse* occurs and the *Failed-Method-Impasse* process is engaged for reflecting down another method `M'`.

Then, *Noos-Eval* and *Failed-Method-Impasse* processes are iterated until a method succeeds or there are no more methods to reflect down. If the evaluation of a method succeeds, task `F(D)` is achieved yielding the value inferred in that evaluation; otherwise *Next-Value* fails.

Failed-Method-Impasse

The *Failed-Method-Impasse* process is engaged when the evaluation of a method `M` fails in solving a task `F(D)`. Then, the control of the inference is passed to the metalevel. Being `ML` the metalevel of `D`, and $\{M_i\}_{F(D)}$ the set of partially ordered alternative methods for solving task `F(D)` already inferred by metalevel task `F(ML)`, the goal of the *Failed-Method-Impasse* process is to select a not previously selected method, from the alternative set of methods $\{M_i\}_{F(D)}$, for solving task `F(D)`.

When all alternative methods have already been selected and have failed, the *Next-Value* process is engaged at the metalevel for the metalevel task `F(ML)` engaging backtracking in the metalevel method defined in `F(ML)`. If *Failed-Method-Impasse* fails, then no method M_i, inferrable at the metalevel `ML`, that solves `F(D)` given current engagements.

Failed-Method-Impasse process involves two different actions:

Given an impasse generated in solving a task `F(D)`, being `ML` the metalevel term of `D`, and being $\{M_i\}_{F(D)}$ the set of partially ordered alternative methods for solving task `F(D)` already inferred by `F(ML)`,

1. When there are alternative methods not previously selected, a non-failed method `M`, maximal with respect to preference, is selected. Then, the selected method is reflected down to task `F(D)` reinstantiating method `M` in the context of task `F(D)`.

2. When all alternative methods $\{M_i\}_{F(D)}$ have already been selected and have failed, the *Next-Value* process is engaged for the metalevel task `F(ML)`.

   (a) When metalevel task `F(ML)` is achieved, it yields a set of partially ordered alternative methods $\{M_i'\}_{F(D)}$ for solving task `F(D)`. Otherwise the *Failed-Method-Impasse* process fails.

   (b) Then, one of the maximal methods $\mathtt{M}_j$, from $\{M_i'\}_{F(D)}$, is selected according to the preference.

   (c) Finally, the selected method $\mathtt{M}_j$ is reflected down to task `F(D)` reinstantiating $\mathtt{M}_j$ in the context of task `F(D)`.

After the impasse is resolved, object-level inference is resumed.

Eval-Next

The *Eval-Next* process is engaged when backtracking is forced in the evaluation of method `M`. If *Eval-Next* fails no other value, resulting from the evaluation of `M`, exists given current engagements.

Specifically, given a method M that is a refinement of a built-in method B with required features F_1, F_2, ..., F_n and tasks F_1(M), F_2(M), ..., F_n(M) engaged for inferring the value of the required features,

1. Backtracking is engaged in task F_n(M) using the *Next-Value* process.
2. Backtracking on task F_n(M) can succeed or can fail: when it succeeds, the evaluation of method M is resumed to next step; when it fails, backtracking on task F_{n-1}(M) is recursively engaged (see *Noos-Eval* process). When backtracking on the first subtask F_1(M) fails, the *Eval-Next* process fails.
3. When all tasks F_1(M), F_2(M), ..., F_n(M) are achieved, a specific combination of the results of the subtasks, according to the built-in definition of B, is performed and the method yields this value.

The *Eval-Next* backtracking process assures that all the possible collections of values for tasks F_1(M), F_2(M), ..., F_n(M) have been tried before determining the failure in the evaluation of a method.

Moreover, at the end of the inference of an achieved problem task, the collection 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 3.1 ***(Maximal solution)*** *Given the set of achieved subtasks $t_1, t_2 \cdots t_n$, that form the task decomposition of a problem task F(D), 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 F(D) 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 F(D).*

The definition just given is indeterministic when the maximal is not unique, and corresponds to the formalization developed in Section 5.13. In order to avoid this indeterminism, the Noos inference engine implementation determines a pre-established execution order among the subtasks of a method, and when no preference order can be inferred between a set of alternative methods for solving a specific task, the writing order is used for determining a total order.

3.5.5 An example of inference

Let us to show the Noos inference process using a short example. Suppose we define a concept `stuff` with a feature `kind` determining whether a given stuff is `material` or `ideal` using two methods as follows:

```
(define (any stuff)
  ((kind (define (conditional)
           (condition (>> solid?  made-of))
           (result material))
```

```
(define (conditional)
   (condition (>> spiritual?  made-of))
   (result ideal)))))
```

Then, we define a concept `spiritual-stuff` by refinement of `stuff` as follows:

```
(define (stuff spiritual-stuff)
  (made-of (define (thing)
             (spiritual?  true))))
```

Next, using the following query expression

```
(>> kind of spiritual-stuff)
```

the *Engage-Task* process starts for problem task `kind(spiritual-stuff)`. Figure 3.20 shows the trace of the inference engaged in Noos in this short example. We will describe the inference steps below indicating the line number from the trace given in Figure 3.20:

(3) The *Engage-Task* process for problem task `kind(spiritual-stuff)` is engaged.

(4) Since there is no method for feature `kind` specified in `spiritual-stuff`, an impasse occurs and the control of the inference is passed to the *No-Method-Impasse* process.

(6) The *No-Method-Impasse* process engages first a metalevel task for obtaining the set of alternative methods for solving task `kind(spiritual-stuff)`.

(7) Two alternative methods, defined at the metalevel of `stuff`, and that we will call `conditional-1` and `conditional-2` are yielded.

(8) First `conditional-1` is selected and reflected down to the baselevel `spiritual-stuff`.

(10) Next, the *Noos-eval* process is engaged for evaluating the method reflected down (`conditional-1'`).

(12) The evaluation of the method performs first the subtask `condition` that, in turn, engages the evaluation of the path reference `(>> solid? made-of)` in the scope of `spiritual-stuff`.

(16) Since the path reference cannot be reduced, *Noos-eval* process for `conditional-1'` fails.

(17) This failure engages a new impasse and the control of the inference is passed to the *Failed-Method-Impasse* process.

(19) Then, the next method `conditional-2` is selected and reflected down to `spiritual-stuff`.

```
(>> kind of spiritual-stuff)

Eval: <Infer-value (>> kind of spiritual-stuff)>
⟹
   Task: kind(<spiritual-stuff>)
   Impasse:  No-method
   ⟹
     Task: kind(<meta of spiritual-stuff>)
     Value: <set of <conditional-1> <conditional-2>>
     Select <conditional-1>
   ⟸
   Eval: <conditional-1'>
   ⟹
     Task: condition(<conditional-1'>)
     Eval: <Infer-value (>> solid?  made-of)>
     %FAIL%
   ⟸
   %FAIL%
   Impasse:  Failed-method
   ⟹
     Select <conditional-2>
   ⟸
   Eval: <conditional-2'>
   ⟹
     Task: condition(<conditional-2'>)
     Eval: <Infer-value (>> spiritual?  made-of)>
     Value: <true>
     Task: result(<conditional-2'>)
     Value: <ideal>
   ⟸
   Value: <ideal>
⟸
Value: <ideal>

<ideal>
```

Figure 3.20. Inference trace.

(21) Next, the the *Noos-eval* process is engaged for evaluating the method reflected down (`conditional-2'`).

(23) The evaluation of `conditional-2'` performs first the subtask `condition` evaluating, in turn, the path reference `(>> spiritual? made-of)` in the scope of `spiritual-stuff`.

(29) Since the result of the subtask `condition` is `true`, the result returned by the method is the constant feature value defined in subtask `result`, that is to say, `ideal`.

Finally, `ideal` is returned as the solution for the query expression.

3.6 Summary

In this chapter we presented the different elements of the Noos representation language. Since we introduced many concepts of the language, now we will summarize the main features of Noos.

In the first section, we described the Noos modeling framework based on four knowledge categories: domain knowledge, problem solving knowledge, episodic knowledge, and metalevel knowledge. Then, we presented how these four knowledge categories are represented in Noos introducing incrementally the different elements of the Noos language.

First, we described the basic elements of the language:

- We presented *descriptions*, the syntax Noos uses for constructing feature terms. A description clusters together as a collection of features the relations in which a concept is involved. Features are interpreted as functions over sets. This view allows to define several feature terms as the value of a feature.

- Next, we introduced *refinement*, an operation for constructing feature terms that involves two distinct aspects: (1) code reuse (the construction of a feature term by reusing another feature term) and, (2) subtyping (the definition of a domain-specific sort hierarchy).

- Then, two forms of reference are presented: *name reference* and *path reference*. Name references are used for defining feature values by referring to feature terms defined elsewhere. Path references are used for designating any feature term `F` by specifying a sequence of feature names that from a feature term `F'` leads to `F`. Path references define *path equalities* between features that can be seen as constraints. Moreover, the usual interpretation of path references is enlarged for allowing path references to deal with feature values that are sets.

- The last basic element of Noos are *methods*. Methods are represented as evaluable feature terms and are also constructed by descriptions. The set of features defined in a method description is interpreted either as a reference to some knowledge source required by the method, or as a subtask required to be accomplished by the method. Noos provides a set of built-in methods and new methods can be defined from them by refinement.

The uniform representation of methods in Noos allows to represent problem solving knowledge in the same formalism. Thus, our approach provides simpler representation constructs than other hybrid representation languages such as LOOM [MacGregor, 1994] [MacGregor, 1991] and CARIN [Levy and Rousset, 1996]. These systems combines a description logic representation with a datalog like rule language for representing problem solving knowledge. A similar approach is taken in the CLASSIC system [Brachman et al., 1991] where forward-chaining rules are integrated with a description logic representation as an added constructor.

Next, we presented the reflective capabilities of Noos introducing the remainder elements of Noos. Three metalevel components are defined in Noos for representing the metalevel knowledge: metalevels, default metalevels, and tasks. All of them are represented uniformly as feature terms.

- A metalevel contains knowledge about a concept, called *referent*, together with the collection of methods that are applicable to each feature of its referent. Using metalevels, multiple methods can be defined to achieve a task. Metalevel methods can also be defined in the features of a metalevel for dynamically computing a set of ordered methods for solving a task taking into account the information available in the current problem.

- A default metalevel is a special kind of metalevel that contains a set of methods that can be applied to all the features of its referent.

- Tasks reify the status of the inference in the language. Tasks embody knowledge such as the method that has succeeded in achieving that task and the result of the evaluation of the method.

A collection of reflective operations defined in Noos allows to access to and to inspect all the metalevel components (namely `meta`, `default`, `task`, `current-task`, and `referent`). As we will present in the next chapter, reflective operations are a basic component for the integration of learning and problem solving in Noos.

Another element introduced in this chapter is the notion of *reification*. Reification allows to express the inference process involved in reducing path references as Noos methods. Path references are reified as *query-methods*. Using query-methods, a set of four different metalevel inference capabilities about feature values is provided (namely `infer-value`, `exists-value`, `known-value`, and `all-values`). The inference process involved in the evaluation of methods is also reified as Noos methods. Analogously to path references, the reification of the evaluation of methods provides a set of metalevel inference capabilities about method evaluation (namely `noos-eval`, `exists-eval`, `known-eval`, and `all-eval`).

The automatic reinstantiation mechanism of Noos was also presented. Reinstantiation is a powerful mechanism for integrating learning that allows, as we will show in the next chapter, to support derivational analogy reasoning in Noos.

Then, we described *preferences*, a declarative mechanism for decision making about sets of alternatives. Preferences are a declarative control mechanism for determining the order in which a metalevel task chooses a method for a task from a set of alternative methods. Furthermore, preferences are used in Noos as a symbolic representation of relevance in comparing a given current problem with problems previously solved by the system. Specifically, preferences are used in the retrieval and selection of precedent cases in case-based reasoning. Different examples of the use of preferences in Noos are given in Chapter 6.

Finally, we described how inference is performed in Noos. We introduced the notion of *impasse* and *backtracking*.

When solving a task where neither a path reference nor a method is defined, an impasse occurs and the control of the inference is passed to its corresponding metalevel task. Solving an impasse for a task `F(D)` involves three processes: (i) determining a set of methods $\{M_i\}_{F(D)}$ applicable to task `F(D)`, that can be partially ordered with preferences, (ii) selecting a method from $\{M_i\}_{F(D)}$, according to the preferences, and (iii) reflecting down the selected method to task `F(D)`.

Backtracking is engaged when a method fails in solving a task. In that case, another remaining non-failed method in $\{M_i\}_{F(D)}$ will be selected and reflected down. Moreover, since a method `M` can have subtasks, and each subtask may have several alternative methods to solve it, metalevel inference ensures that backtracking is engaged in `M`. Then, the possible combinations of methods for each subtask are tried, following the local preference orderings for each subtask, until a solution is found.

Next Chapter will discuss the role of experience and memory in Noos problem solving and our proposal for integrating learning techniques in Noos. In the next Chapter we will present the Noos elements concerning to the integration of learning. The reader can find in Appendix A the rest of elements provided in the Noos development environment.