Chapter 4

Memory, Experience and Learning

It is clear the importance of experience's role in human problem solving. Experience allows people to learn how to focus on relevant details of a problem, to avoid decisions that have previously resulted in failure situations, etc. The incorporation of experience capabilities in knowledge systems also plays an important role in order to improve their behavior. As it was argued in [Kolodner and Riesbek, 1986], reasoning about the experience in problem solving involves two main aspects:

- a *memory structure* that stores the decisions taken by the system in the solution of problems, and
- the capability of inspecting, retrieving and reasoning about these decisions.

Another important remark is that, because memory changes depending of the experience, the results of asking the system to solve a given task at two different times may be different.

We call the memory structure that stores the decisions taken by Noos in the solution of problems *episodic memory*. The set of decisions stored in the episodic memory form the *episodic knowledge* of the system (see Section 3.1). Episodic knowledge holds information and decisions used in solving particular *episodes* (particular problems). This kind of knowledge requires the system to have a model of certain aspects of itself (a *self-model*). The type of self-model is determinant to the kind of learning that can be performed in the system.

In this chapter we will present the components of the episodic knowledge that constitute the *episodic memory* of Noos. We will also present three mechanisms for inspecting the episodic memory: *access by path*, that provides an access to the episodic memory combining reflective operations and path references; *retrieval methods*, that provide a powerful mechanism for accessing to the episodic memory contents; and *perspectives*, a mechanism to describe declarative biases for case retrieval in structured representations of cases. Next, we will deal with the role of learning and its integration into the Noos language. Finally, we will explain how different symbolic learning approaches—such as case-based reasoning, inductive learning, and analytical learning—are incorporated in Noos.

4.1 Episodic knowledge in Noos

Episodic knowledge in Noos is represented as the set of tasks, methods, preferences, and problem data involved in solving problems. Episodic knowledge is organized in episodic models. Each episodic model holds the reification (the self-model) of the inference process engaged in Noos in solving a specific problem task.

Episodic model

An episodic model is the *explanation* of the inference process engaged by Noos in solving a specific problem task. In computational terms, it could be said that it is the trace of the program that solves a specific problem task.

An episodic model holds the set of knowledge pieces used for solving a specific problem task, how and where they were used, and the decisions taken for solving that problem. Specifically, the episodic model constructed by Noos for solving a specific problem task `F(D)` holds the problem data, the solution for that problem, and the problem task engaged for solving that problem. This problem task `F(D)` is reified as a task feature term (see Section 3.3.4) and holds, in turn, the following knowledge:

- the name of the task (that is the name of the feature `F`),
- the feature term to which the task is addressed (`D`),
- the solution inferred by the task, and
- the method `M` that has succeeded in achieving that task.

The method `M`, in turn, holds:

- the values inferred by its subtasks (the feature values of `M`'s features),
- the task feature terms of `M` (i.e. the reification of the subtasks of `M`).

When a task engaged in problem solving has no method or has multiple alternative methods for achieving that task, a metalevel task is engaged by Noos (see Section 3.5). These metalevel tasks are also tasks that are part of the episodic model. A metalevel task `MT` is a task that has a name that is the same name as the name of the task `T` it solves, is addressed to the metalevel term to which the task `T` is addressed, holds a metalevel method, and holds the solution inferred by `MT`—a set of partially ordered alternative methods for solving the task `T`.

Summarizing, the episodic model constructed by Noos for solving a specific problem task `F(D)` holds the problem data given for solving `F(D)` and the task/method decomposition tree engaged in solving F(D); this tree, in turn, holds the methods succeeded in achieving each task, the reification of subtasks engaged by each method, and the reification of the metalevel tasks that have been engaged.

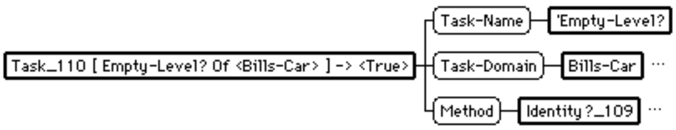
Once a problem task is solved Noos automatically memorizes (stores and indexes) the episodic model that has been built.

Let us introduce an example of the episodic model built in solving a specific problem task. Given the following definition of a car:

```
(define Bills-car
  (owner Bill)
  (gas-level-in-tank 2)
  ((gas-gauge-reading (define (conditional)
                        ((condition (define (lower-than?)
                                       (is-lower (>> gas-level-in-tank))
                                       (than 5))))
                        (result empty)
                        (otherwise full))))
  ((empty-level?  (define (Identity?)
                     (item1 empty)
                     (item2 (>> gas-gauge-reading))))))
```

and solving the problem task empty-level?(Bills-car), the identity method is evaluated requiring the value of feature gas-gauge-reading. In turn, the conditional method defined in the gas-gauge-reading feature and the lower-than? method defined in its condition subtask are evaluated yielding empty as result (since the value of feature gas-level-in-tank is lower than 5). Finally, the solution yielded for the problem task is true.

The episodic model built for that problem task holds a task with name empty-level?, addressed to Bills-car feature term, with solution value true, and with an identity method with printname <identity?_109> as following:



The <identity?_109> method has, in turn, two subtasks (see Figure 4.1): item1 and item2 subtasks with their corresponding values. Task item1 has a constant value empty. Task item2 holds an infer-value method that is the reification of the path reference (>> gas-gauge-reading) in the scope of Bills-car. This infer-value method has, in turn, three subtasks: feature, holding the constant value gas-gauge-reading; domain, holding the constant value Bills-car; and the task gas-gauge-reading(Bills-car) engaged for solving the feature value for feature gas-gauge-reading of Bills-car. This task holds the conditional method <conditional_110>. The <conditional_110> method has, in turn, two subtasks: condition, holding a numerical comparison method <lower-than?_112>; and result, holding the content value empty. The comparison method has, in turn, two tasks: is-lower and than. Task is-lower

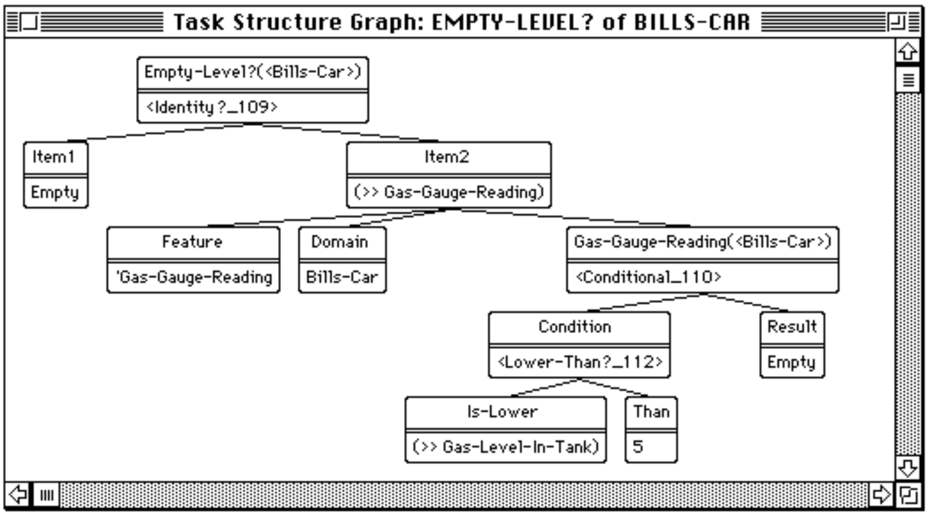


Figure 4.1. A browser of the task/method decomposition from the episodic model of the `empty-level?(Bills-car)` problem task.

has another infer-value method that is the reification of the path reference `(>> gas-level-in-tank)` in the scope of `Bills-car`. Finally, task `than` has the constant value `5`.

All these are components of the episodic model of the problem task `empty-level?(Bills-car)` and are accessible and inspectable as we will presently explain.

Episodic memory

Episodic memory is the collection of the episodic models of the problem tasks that the system has solved.

Noos provides three ways of accessing and reusing episodic models for solving new problems. That is, three ways to examine the contents of episodic memory. The first one is an *access by path*. The second one is an *access by contents*. The third one are *perspectives*.

Access by path is performed by combining reflective operations and path references (see Section 3.3.5). Access by path provides a way to access specific portions of the episodic memory. For instance, we can access to the method that solved the problem task `empty-level?(Bills-car)` using the following combination:

```
(>> method of (task empty-level?  of Bills-car))
```

yielding `<identity?_109>`.

Since methods are feature terms, they are inspectable to external methods in a uniform manner to other feature terms. Parameters and subtasks of a method are features. Thus, they can be accessed using a path. For instance, we can access the value of the `item1` subtask of `<identity?_109>` method as follows,

```
(>> item1 method of (task empty-level?  of Bills-car))
```

yielding `empty`, or the method of the `item2` subtask of `<identity?_109>` method as follows,

```
(>> method of (task item2 of
                    (>> method of (task empty-level?  of Bills-car))))
```

yielding method `<infer-value (>> gas-gauge-reading)>`.

We can say thus that Noos methods are *transparent*. The transparent capability of Noos methods allows to perform forms of inference that need to inspect and reason about methods and how they have been used to solve particular tasks.

An example of using introspection over methods is analytical learning (see Section 4.7). An analytical learning method builds more efficient and compact methods by examining the explanation (the episodic model) of the methods used to solve a specific problem task.

On the other side, *access by contents* is performed by retrieval methods. Retrieval methods provide a way to retrieve parts of the episodic memory using the notion of feature terms as partial descriptions and the subsumption ordering among them.

4.2 Retrieval

Noos provides a set of basic retrieval methods. Retrieval methods allow to retrieve previous relevant episodes from the episodic memory using relevance criteria. Relevance criteria are determined by specific domain knowledge about the importance of different features or by requirements of problem solving methods.

Usually, the notion of similitude in case-based reasoning introduces a way to assess the relevance of precedent cases in solving a new case. Similarity measures estimate a relevance order between precedent cases. Our approach is to work directly over relevance orders.

Retrieval methods are based on the notion of feature terms as partial descriptions and the notion of subsumption among feature terms (see Chapter 5). The intuitive meaning of subsumption is that a term t_1 subsumes another term t_2 ($t_1 \sqsubseteq t_2$) when all information in t_1 is also contained in t_2. Our approach is that a knowledge modeling analysis can determine the relevant aspects of problems; then, partial descriptions of the current problem can be built embodying the aspects considered as relevant. These partial descriptions are used as retrieval patterns for searching similar cases in the episodic memory using subsumption. Thus, retrieval methods can be viewed as methods that search into the episodic memory the set of feature terms subsumed by a feature term, a pattern, embodying the relevant aspects of a problem data.

For instance, in the diagnosis of car malfunctions domain, we could be interested in solving the diagnosis task of a new car by means of looking to previous solved diagnosed cars with the same symptoms as that of the new car. That is to say, retrieving feature terms of sort `car`, with same feature value for feature `symptom` as the new car, and with feature value for feature `diagnosis` a feature term of sort `malfunction`. Given a specific car with feature value `does-not-start` for the `symptom` feature, the feature term embodying this partial description can be defined as follows:

```
(define (Car)
  (symptom does-not-start)
  (diagnosis malfunction))
```

This retrieval mechanism is achieved by the `retrieve-by-pattern` built-in method. The `retrieve-by-pattern` built-in method has a required feature called `pattern`. The feature value of `pattern` is taken as the subsumer feature term used for retrieval over the episodic memory. For instance, we may define a retrieval method for the diagnosis of car malfunctions domain as follows:

```
(define (retrieve-by-pattern Search-diag-cars)
  (symptom complaint)
  (pattern (define (Car)
             (symptom (>> symptom))
             (diagnosis malfunction)))
```

where we are defining a retrieval method, called `Search-diag-cars` method, by refinement of the `retrieve-by-pattern` method, and with a parameter called `symptom`.

The evaluation of the `Search-diag-cars` method in a specific car with a specific `complaint` performs a search into the episodic memory yielding as result the set of cars previously diagnosed by the system (i.e. that have some diagnosis of sort malfunction) and with that specific `complaint` as feature value of their `symptom` feature.

As we have shown, since retrieval methods are methods like any other built-in method provided in Noos, new retrieval methods can be designed refining and combining the existing ones from a knowledge modeling analysis of problems.

Retrieval by contents is a powerful capability that allows to develop methods based on analogical reasoning. For instance, analogy by determination [Russell, 1990] may be directly performed as a *retrieve-by-pattern* method. The approach of analogy by determination applied in solving a problem $P(A)$, given the information that $P(A)$ is determined by $Q(A)$ and given an example of another solved problem $P(B)$ such that $Q(A) = Q(B)$, is that problem $P(A)$ can be resolved as $P(A) = P(B)$.

For instance, a classical example of an analogy justified by a determination is that the usual language spoken by a person is determined by the person's nationality. In Noos this can be easily performed by the following method:

```
(define (decomposition Language-determination)
  (citizen person)
  ((precedents (define (retrieve-by-pattern)
                 (pattern (define (person)
                            (national-of (>> national-of citizen))))))
  (language (>> speaks precedents)))
```

that can be used for solving the `speaks` feature of a `person` as follows:

```
(define Person
  (national-of country)
  ((speaks (define (language-determination)
             (citizen (>>))))))
```

Then, given an example of a person called `Janos` that is `Hungarian` and speaks `Magyar`,

```
(define (Person Janos)
  (national-of Hungary)
  (speaks Magyar))
```

and given the `speaks` task of another person that is also `Hungarian`,

```
(define (Person Petia)
  (national-of Hungary))
```

we can conclude that `Petia` speaks `Magyar`.

In Section 4.5 we provide an example of the use of a retrieval method for acquiring methods in an analogical reasoning approach. In Appendix D we describe all built-in retrieval methods.

4.3 Perspectives

Perspectives is a clear and flexible mechanism to describe declarative biases for retrieval in the Noos episodic memory. Our approach is based on the observation that, in complex tasks, the identification of the relevant aspects for retrieval in a given situation may involve the use of knowledge intensive methods. This identification process requires dynamical decisions about the relevant aspects of a problem and involves introspection.

The view of feature terms as partial descriptions allows the representation of declarative biases also as feature terms in a natural way. The declarative biases are interpreted as syntactic patterns. *Perspectives* are the way to construct, from these syntactic patterns, partial descriptions of the current problem embodying only those aspects considered relevant. These partial descriptions may be used later for retrieval. Figure 4.2 shows the use of perspectives into a retrieval task. The formal description of perspectives is presented in Section 5.11.

Perspectives are constructed in Noos using the `perspective` built-in method. The `perspective` built-in method has two required features called `pattern` and `source`. The feature value of the `pattern` feature is taken as the syntactic

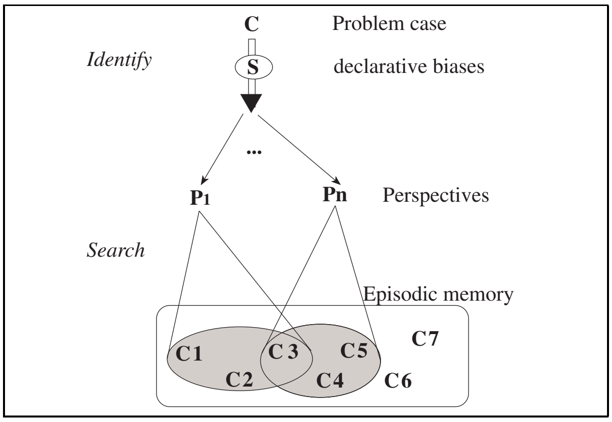


Figure 4.2. Using perspectives in a retrieval task. First, given a problem case C and using syntactic patterns S as declarative biases, the relevant aspects of C are determined and perspectives $P_1 \cdots P_n$ are built. Later, these perspectives are used in retrieval methods to search precedent cases in the episodic memory.

pattern that describes, as a declarative bias, the relevant aspects of a problem situation (a feature term) given in the `source` feature. The `perspective` method constructs a new term extracting from the `source` term those aspects declared in the syntactic pattern given in `pattern`.

There are two possibilities for constructing perspectives. The first option is to use a syntactic pattern where the relevance of some feature values is declared using sorts. For instance, we will take an example from the *Saxex* musical application (see Section 6.5 for a detailed description). Let as consider a note `Note1` defined as follows:

```
(define (note Note1)
  (pitch C5)
  (position 0)
  (duration Q)
  (metrical-strength extremely-high)
  (belongs-to (define (P)
               (first-note (>>))
               (med-notes Note2)
               (last-note Note3)
               (direction down)))
  (next Note2))
```

Moreover, let us consider as relevant aspects of a note its duration and its metrical strength on the melody. We can specify this knowledge by means of the use of the following syntactic pattern in a perspective method `P1` for constructing a perspective of `note1` (defined before) as follows:

```
(define (perspective P1)
  (source Note1)
  (pattern (define (note)
             (duration rhythm)
             (metrical-strength strength))))
```

the following perspective will be constructed by `P1`,

```
(define (note)
  (duration Q)
  (metrical-strength extremely-high))
```

This term, in turn, can be used for retrieval obtaining, as a result, the set of "note precedents" from the episodic memory with duration Q (quarter duration) and a *extremely-high* metrical strength.

The second way to build a perspective is to use a syntactic pattern where the relevance of some features is declared using *variables of features*. This alternative allows to identify the roles of features and terms in the structure. For instance, still in the *Saxex* application, let us to consider as a relevant aspect of a note the "role" that plays in the analysis structure of the musical phrase it belongs. Using the cognition model of musical understanding of Narmour's theory (see Section 6.5 for more details) a phrase can be analyzed by grouping the notes in a sequence of basic structures (that we model by refining the common sort `N-structure`). A structure assigns a different role to each note (that we represent with features) belonging to the structure. Since the feature names differ, we can specify this knowledge by means of using variables of features; we can use the following syntactic pattern, where feature variables are noted with the \$ symbol, in a perspective method `P2` for constructing a perspective of `note1` as follows:

```
(define (perspective P2)
  (source Note1)
  (pattern (define (note)
             (belongs-to (define (N-structure)
                           ($f (>> pattern)))))))
```

Specifically, since `Note1` is the `first-note` of a melodic `P` process structure (`P` is a kind of `N-structure`), according to Narmour's theory, the following perspective will be constructed:

```
(define (note)
  (belongs-to (define (P)
                (first-note (>>)))))
```

Finally, using this perspective for retrieval we obtain, from the memory of cases, all the notes playing the same role that `Note1` (i.e. first notes of melodic process structures).

Using a specific syntactic pattern multiple perspectives can be constructed. There are two factors that allow the construction of diverse perspectives: the specification of a variable of features and the specification of sets in feature values. The backtracking mechanism of the Noos inference engine allows to obtain all the perspectives consecutively.

4.4 Reasoning and learning

Machine Learning (ML) techniques have been used by knowledge modeling methodologies as a way to acquire certain models in the knowledge acquisition (KA) process conducive to building a knowledge system. Our interest is in developing knowledge systems with *integrated* learning capabilities: we are interested in developing different machine learning methods and integrate them into the problem solving of knowledge systems. This option means essentially that certain knowledge acquisition tasks are *delayed* from the knowledge system design and construction phase to the phase in which the knowledge system is actually used in the task environment. Since knowledge modeling methodologies view KA as a process that basically build models, our approach means that some models are not built[1] in the first phase, and their construction is delayed to the second phase where appropriate ML methods are appointed to generate those models.

This delay of KA tasks also implies the following:

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

Moreover, our approach is that any time some knowledge is required by a problem solving method, and that knowledge is not directly available, there is an opportunity for learning. Thus, our proposal is the use of learning methods whose task is the acquisition of this lacking knowledge in some problem solving process.

Moreover, our proposal is that learning methods are methods with introspection capabilities that can be analyzed also by means of a task/method decomposition. For instance, case-based reasoning methods require access to precedently solved similar problems (called cases), select some of them using some criteria, and finally adapt the solutions from the cases to the current problem. This adaptation phase can be just to use the exactly same previous solutions, to re-instantiate the solutions in the new problem situation, or to construct a new solution according the previous solutions and the current problem. As a second example, inductive learning methods need a set of examples in order to construct a new description that characterizes a target concept. Explanation-based

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

learning methods analyze inference processes and construct new knowledge that is incorporated as a new domain theory, or strategic knowledge that will be used in the resolution of new problems.

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

Introspection This task is the process by which past experience (episodic memory of the system itself or provided by a teacher) is accessed, selected and retrieved for the purpose of solving new problems. In simple situations this task may merely select a subset of examples in memory. In complex situations the system may have to decide which (sub)parts of all the episodic memory qualify as "examples", i.e. they are interesting to learn from (see Section 4.3).

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

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

Construction task involves methods that usually are called *learning algorithms*. This is because in off-line learning the "introspection" task is simply done by a human engineering the system, while revision is present only in incremental algorithms—and in interactive systems decisions about revision are taken by the human engineering the system. These human-intensive processes are modeled in our framework as methods for the introspection and revision tasks that may interact or not with the human expert/engineer.

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

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

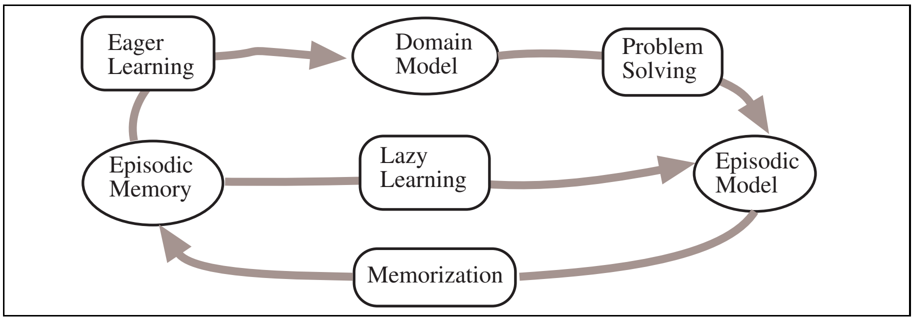


Figure 4.3. Lazy and eager learning and the construction of domain models and episodic models.

Lazy learning Past experience is accessed, selected and used in a problem-centered approach. The paradigmatic example is CBR, where for each new problem the system filters out irrelevant past experiences, and focuses on the relevant part from which it extracts or generates new knowledge to the extend needed for solving that particular new problem. We view lazy learning as constructing an *episodic model* for the current problem—instead of constructing a generic model[2].

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

4.5 Case-Based Reasoning

Case-based reasoning (CBR) forms a family of techniques and systems that integrate lazy learning with problem solving where domain-specific knowledge and methods are used. We model CBR in Noos as *case-based methods*. It is clear that case-based methods can be integrated in our framework because of the notion of memory: past problem solving episodes (episodic models) are stored in memory and can be recalled using the retrieval methods. These stored problem solving episodes constitute the set of precedents (also called cases) for

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

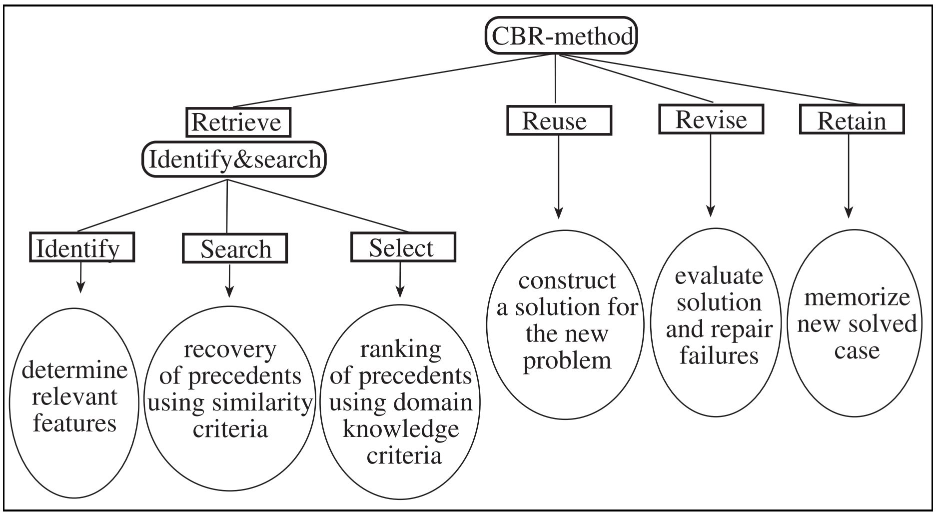


Figure 4.4. Task/method decomposition of Case-based reasoning methods.

case-based methods. Generally speaking, case-based methods are decomposed into four subtasks [Aamodt and Plaza, 1994]: `retrieve`, `reuse`, `revise`, and `retain` (see Figure 4.4). Since there are several possible methods usable in each subtask, several CBR techniques can be integrated in this way. Next we will describe the goal of each CBR subtask indicating how can be achieved using Noos methods:

- `Retrieve`: the `retrieve` subtask requires a method that recovers previous solved cases, from the episodic memory, "similar" to the current problem. A retrieval method is usually decomposed in three (sub)tasks: `identify`, `search`, and `select` tasks.
  - `Identify`: the goal of the `identify` task is to determine a set of relevant aspects of the current problem using knowledge about the problem to be solved. In Noos the identification task is mainly performed using perspectives that determine relevance criteria.
  - `Search`: the goal of the `search` task is to look for precedent cases in the episodic memory using the relevance criteria constructed by an identification method. The `search` task is performed in Noos using retrieval methods.
  - `Select`: the goal of the `select` task is to rank the cases obtained in the previous task according to domain criteria. The `select` task is performed in Noos combining preference methods. The `select` task can choose the most relevant case or an ordered set (that can be partial) of relevant cases.

- `Reuse`: given a set of relevant cases, the goal of the `reuse` task is to build a solution for the current problem adapting the solutions given in the cases. Usually in CBR the solution is built either taking the solution given in the most relevant precedent or constructing a new solution by adapting the solution(s) of one or more precedents. An usual method for `reuse` subtask is what Carbonell called *derivational analogy* [Carbonell, 1986] (see Section 4.5.1). Another alternative is to use a set of transformation operations. Transformation operations can be implemented providing methods that access to solutions used in precedents and constructs, using domain specific knowledge, a new solution.

- `Revise`: When the solution built by the `reuse` task is not correct, an opportunity for learning arises. The `revise` task usually involves two subtasks: the error detection task and the repair task. The error detection task is usually a task performed outside of the CBR system. A possible way to implement it in Noos is asking to the user using the set of methods provided in Noos for interacting with users. The repair task is usually implemented using causal knowledge for generating an explanation of why certain parts of the solution case were not achieved. Repair methods can be built in Noos as adapting methods in a similar way that methods constructed for `reuse` task.

- `Retain`: the goal of the retain task is to incorporate the new solved problem to the memory of cases in order to help the resolution of future problems. The incorporation of the new solved problem to the episodic memory is performed automatically in Noos. All solved problems, in principle, may be available for the reasoning process in future problems. Noos programming environment provides a way to explicitly determine which solved problems have to be stored (see Appendix A).

4.5.1 Derivational analogy

Derivational analogy [Carbonell, 1986] is a powerful reasoning mechanism for transferring knowledge from past episodic models to a new situation, based on preserving decisions that apply in the new situation and replacing or modifying those that are no longer valid in the new situation.

Derivational analogy is automatically supported by Noos using the reinstantiation mechanism (see Section 3.3.7). The reinstantiation mechanism is a metalevel mechanism that provides a way of automatically bind a method in the scope of a new situation using refinement.

Given a current task `F(D)`, and being `MT` the metalevel of `D`, derivational analogy in Noos consists of the following:

1. Once a most relevant precedent case `P` is chosen by the *retrieve* task, the method `M` used in the precedent to solve the same task `F(P)` the system is now involved in is accessed.

```
(define Peters-car
  (complaint does-not-start)
  (battery-voltage low-voltage)
  (gas-level-in-tank full)
  ((diagnosis (define (Causal-Explanation)
              ((cause (define (Identity?)
                        (item1 low-voltage)
                        (item2 (>> battery-voltage)))))
              (effect low-battery-malfunction))))
  ((empty-level?  (define (Identity?)
                  (item1 empty)
                  (item2 (>> gas-level-in-tank))))))

(define Carols-car
  (complaint does-not-start)
  (battery-voltage high-voltage)
  (gas-gauge-reading empty)
  ((diagnosis (define (Causal-Explanation)
              (cause (>> empty-level?))
              (effect no-gas-malfunction))))
  ((empty-level?  (define (Identity?)
                  (item1 empty)
                  (item2 (>> gas-gauge-reading))))))
```

Figure 4.5. Two precedents from the episodic memory of solved diagnosed cars.

2. The method is re-instantiated to the current problem—i.e. method references are mapped from the past case `P` to the current case `D` and bound in the scope of `D`.

3. The new method is used to solve the current task `F(D)`.

Let us introduce a short example of a metalevel method called `basic-analogy`, a case-based method for learning methods that succeeded in achieving a given problem task in past precedents. The `basic-analogy` method is decomposed in two subtasks, `retrieve` and `reuse`, as follows:

```
(define (sequence basic-analogy)
  ((retrieve (define (retrieve-by-task)
             (task-name (>> task-name of (current-task)))))
  (reuse (>> method of (task (>> task-name of (current-task)) of
                          (>> precedents)))))
```

The goal of the `retrieve` task is to search into the episodic memory the set of precedent cases that solved the same task the system is involved (that task is accessed using the `current-task` reflective operator). Given a set of precedents, the `reuse` subtask takes the methods applied in that precedents for solving the current task. The Noos inference engine assures that all of them will be re-instantiated successively to the current problem until one of them success in achieving the task.

Note that the `basic-analogy` method is an introspective method that uses the self-model of Noos for determining the task at hand and accesses to episodic model components, such as tasks and methods used in solving those tasks.

The `basic-analogy` method can be used in the example of diagnosis of car malfunctions we have already used in Chapter 3. Since the episodic memory of each problem task contains a causal explanation between its complaint and the diagnosis known for that case, an analogical method can retrieve those causal explanations and apply them to a new problem to check which causal explanation also holds in the new problem. For instance, we can define a new problem `Karls-car` where their features, and specifically the `diagnosis` feature, can be achieved including the `basic-analogy` method in the default metalevel of the metalevel of `karls-car` as follows:

```
(define Karls-car
  (complaint does-not-start)
  (battery-voltage high-voltage)
  (gas-level-in-tank empty))

(define (Default (default meta of Karls-car))
  basic-analogy)
```

The `basic-analogy` method retrieves from episodic memory the cases `Peters-car` and `Carols-car` (see Figure 4.5) and then takes the methods, from their episodic models, applied in that precedents for solving the `diagnosis` task. Next, one of them is selected and re-instantiated to the current `Karls-car` problem. Let us suppose that the system select the method applied in `Peters-car` problem. The method retrieved from `Peters-car` fails since that causal explanation does not hold in `karls-car` (the `battery-voltage` is not `low`). Next, the system selects the method applied in `Carols-car` problem.

This method engages in turn the `empty-level?` task. Since there is no method defined in `Karls-car` for that task, `basic-analogy` method is now applied in `empty-level?` task for searching methods for `empty-level?` task on the episodic memory. `Peters-car` and `Carols-car` cases also hold methods for that task and they are retrieved (see Figure 4.5). These methods are reinstantiated in turn and only the method defined in `Peters-car` can be successfully applied to `Karls-car`. Then, the method retrieved from `Carols-car` for `diagnosis` task is resumed and is finally successful, yielding the `no-gas-malfunction` result.

4.6 Inductive learning

The goal of an inductive method is to construct the general knowledge needed by a given problem solving method. Induction performs a generalization from a set of problem solving episodes (usually called 'examples' or 'instances'). In general, the ML community defines induction as a process that constructs, from a set of positive examples and a set of negative examples, a general description that "generalizes" (in some sense that may vary[3]) the positive examples—and does not generalize the negative examples. Induction is also modeled in Noos by methods.

An example of the use of an inductive method is the generation of a class description for a category or concept from a set of examples. The acquired knowledge will be used by an identification method deciding whether or not new examples pertain to a certain category.

In general, inductive methods can be characterized as search methods that follow certain *biases*: constraints over the hypothesis space effectively searched and strategies for searching certain subspaces before others. These bias of ML methods are similar to assumptions for problem solving methods, e.g. a ML inductive method can be exhaustive (or complete—if it assures it will find a generalization if it exists) or not exhaustive. However this comparison is left for future work.

In our framework feature terms offer a representation formalism that is a subset of first order logic. Inductive learning with feature terms is a relational learning [Quinlan, 1990]. Systems that also perform relational learning are ILP systems [Muggleton and De Raedt, 1994]—that uses horn clauses.

Inductive methods currently developed in Noos are based on the *antiunification* and the *subsumption* operations of Noos [Plaza, 1995]. *Subsumption* provides a well defined and natural way for defining generalization relationships: a feature term ψ is more general than another feature term ψ' whenever that ψ subsumes ψ'. The antiunification of two feature terms gives that *which is common to both* (yielding the notion of generalization) and *all that is common to both* (the most specific generalization).

Formally, the antiunification of a set of feature terms yields a greatest lower bound with respect to subsumption ordering.

Definition 4.1 *(Antiunification)*
Given a set of feature terms $\{d_1, d_2, \ldots, d_n\}$ their antiunification is another feature term D such that:

1. *feature terms $\{d_1, d_2, \ldots, d_n\}$ are subsumed by D, and*
2. *there is no feature term D' such that subsumes $\{d_1, d_2, \ldots, d_n\}$ and subsumes D.*

Inductive methods are designed in Noos using antiunification and specific biases. For instance, using relevance measures of attributes we can establish a bias for the generation of disjunctive descriptions of concepts. Several inductive methods based on Noos have been designed, implemented and

[3]Specially in ILP (induction of logic programs) several different semantics have been proposed for the notions of generalization and subsumption (see [Muggleton, 1992] and [Lavrač and Džeroski, 1994] for a detailed discussion).

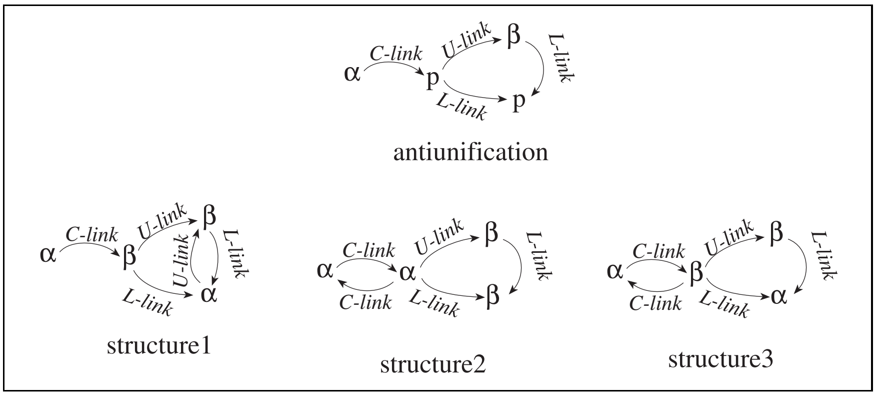


Figure 4.6. Induction example.

tested by Eva Armengol [Armengol, 1997] at the IIIA, and are also described in [Plaza et al., 1996b] and in [Armengol and Plaza, 1997]. In Chapter 6 the use of inductive methods in applications developed in Noos is also presented.

Let us present here a short example of antiunification. Suppose we have a set of examples of chemical structures that are instances of a same chemical concept. Structures are represented as sets of particles of two types (`alpha` particles and `beta` particles) connected among them with three different kinds of features: `U-link`, `C-link`, and `L-link`. For instance, a structure called `structure1` is described as follows:

```
(define (alpha :id structure1)
  (C-link (define (beta)
            (U-link (define (beta)
                      (L-link (>> L-link C-link))))
            (L-link (define (alpha)
                      (U-link (>> U-link C-link)))))))
```

Taking the set of three structures given in Figure 4.6, their antiunification is another structure such that the sort of two particles is generalized (α and β are subsorts of the general particle sort p) and only features common in all examples are preserved.

For the moment Noos inductive methods work only on descriptions and not on methods. What is yet future work is learning of programs (methods) from examples (as in inductive logic programming), although analytical learning of methods has been integrated in Noos as shown in the next section.

4.7 Analytical learning

The goal of analytical learning techniques (or EBL-like learning) is to construct, from a domain theory and a reification of the problem solving process of an example solved using the domain theory, a new domain theory that obeys certain restrictions. Analytical learning techniques are also modeled in Noos by methods.

Specifically, analytical learning is modeled in Noos by methods that given a training example whose problem task `T(E)` has been solved by a problem solving method `M` and given an operationality criterion, construct a new problem solving method $\mathtt{M}_{op}$ for solving task `T` obeying the operationality criterion. The operationality criterion describes a sub-language of the domain model. An operational method will be a method requiring only the knowledge defined by this sub-language.

This learning approach needs to inspect the episodic model built while solving the training example. In Noos the task/method decomposition instantiated in the construction of the episodic model for solving a problem task constitutes the explanation (or trace) of the solution. Analytical learning methods in Noos are metalevel methods that given an episodic model for a task `T(E)` solved by a PSM `M` (1) inspect the methods that succeeded in each subtask of the task/method decomposition tree of `M`, and (2) construct a new PSM for task `T` according to an operational criterion.

We have developed PLEC, an EBL-like learning method that constructs a new operational problem solving method for a specific problem task from a training example and according to a specific operational criterion. The operational criterion of PLEC is that the constructed method refers only to features with constant values in the training example (e.g. those present in `Obj-1` in example below). The result of PLEC is the construction of a new method such that it will be directly applicable to the problem and skip intermediate inferences.

PLEC is a built-in method with two required features: `task-name`, for specifying the name of the task; and `source`, for specifying a training example.

For instance, let us consider a PSM that determines whether or not an object is an example of the `cup` concept defined as follows:

```
(define (conjunction Is-a-cup?)
  (source (define (object)))
  (item1 (>> stable? source))
  ((item2 (define (conjunction)
            (item1 (>> liftable? source))
            (item2 (>> open-vessel? source))))))
```

and let us have the following background knowledge about "cup-like" objects:

```
(define (object)
  (stable? (>> flat? bottom))
  ((liftable? (define (conjunction)
                (item1 (>> graspable?))
                (item2 (>> light?)))))
  (graspable? (>> handle?))
  (open-vessel? (>> upward-pointing? concavity)))
```

Then, a training instance `Obj1` can be defined using the `Is-a-cup?` method in feature `cup?` as follows:

```
(define (object Obj1)
  (owner Fred)
  (light? true)
  (color red)
  (handle? true)
  (bottom (define (part)
            (flat? true)
            (size small)))
  (concavity (define (part)
               (upward-pointing? true)))
  ((cup? (define (Is-a-cup?)
           (source (>>))))))
```

The resolution of the `cup?(Obj1)` problem task by the `Is-a-cup?` method engages in turn other (sub)tasks for determining when a given object is `stable?`, `liftable?`, and `open-vessel?` (see in Figure 4.7 the task/method decomposition hold in the episodic model of the `cup?(Obj1)` problem task). Using the episodic model built after a specific problem is solved, PLEC constructs a new operational method for determining when an object is an example of a cup. Specifically, posing the following query-expression to Noos:

```
(noos-eval (define (PLEC)
             (source Obj1)
             (task-name 'cup?)))
```

The result yielded by PLEC is the following operational method, directly applicable to the problem and skipping intermediate inferences, for the `cup?` task:

```
(define (conjunction Op-Is-a-cup?)
  (source (define (object)))
  ((item1 (define (conjunction)
            (item1 (>> flat? bottom source))
            (item2 (>> handle? source)))))
  ((item2 (define (conjunction)
            (item1 (>> light? source))
            (item2 (>> upward-pointing? concavity source))))))
```

Notice that this new operational method built by PLEC has four path references to features `flat?`, `handle?`, `light?`, and `upward-pointing?` that have constant values in `Obj1`.

4.8 Summary

This chapter presented the components of the episodic knowledge that constitute the *episodic memory* of Noos. Episodic knowledge in Noos is organized in *episodic models*. Each episodic model holds the reification of the inference process engaged in Noos in solving a specific problem task. An episodic model is represented as the set of tasks, methods, preferences, and problem data involved in solving that problem task.

Figure 4.7. A browser of the Task/method decomposition for the `cup?` task of `Obj1`.

All the components of an episodic model are accessible and inspectable. We described in this chapter three different access mechanisms for inspecting episodic models of the episodic memory of Noos: *access by path*, that provides an access to the episodic memory combining reflective operations and path references; *retrieval methods*, that provide a content-based access to the episodic memory; and *perspectives*, a mechanism to describe declarative biases for case retrieval in structured representations of cases.

Since knowledge in Noos is represented in a structured way, retrieval methods have to deal with structured representations. Retrieval methods allow to retrieve previous relevant episodes from the episodic memory using relevance criteria. Relevance criteria are determined by specific domain knowledge about the importance of different features or by requirements of problem solving methods. Retrieval methods are based on the notion of feature terms as partial descriptions and the notion of subsumption among feature terms

Our approach is based on the observation that, in complex tasks, the identification of the relevant aspects for retrieval in a given situation may involve the use of knowledge intensive-methods. This identification process requires dynamical decisions about the relevant aspects of a problem and involves introspection. Perspectives provide Noos with a mechanism for specifying declarative biases. Declarative biases provide a clear and flexible way to express retrieval patterns.

We also presented the role of learning and its integration into the Noos language modeled as introspective methods that can be decomposed of three main subtasks: *Introspection*, *Construction*, and *Revision*.

Finally, we described how three different symbolic learning approaches can be integrated to Noos:

- *Inductive learning methods* are developed in Noos as search methods (that follow certain biases) over the space of feature terms. Inductive learning methods are based on the feature term subsumption and antiunification operations of Noos. Subsumption provides a generalization relationship over feature terms. The antiunification of a set of feature terms builds a new feature term that is a greatest lower bound with respect to the subsumption ordering. Diverse strategies can be developed for constructing inductive learning methods that follow different searching biases.

- *Case-based reasoning methods* are developed in Noos as problem solving methods with lazy learning capabilities that search for previously solved problems in the Noos episodic memory. CBR methods are based on the retrieval and subsumption operations of Noos.
  Structured representations of cases offer the capability of treating subparts of cases as full-fledged cases. That is to say, a new problem can be solved using subparts of multiple cases retrieved from the episodic memory.
  On the other hand, structured representations of cases increase the complexity of retrieval mechanisms. Noos provides elements—such as content-based retrieval and perspectives—for supporting the retrieval on these complex representations of cases.

Furthermore, derivational analogy is automatically supported by the Noos reinstantiation mechanism.

- *Analytical learning methods* are developed in Noos as methods that given a training example whose problem task has been solved by a problem solving method `M` and given an operationality criterion, construct a new problem solving method $\mathtt{M}_{op}$ for solving that task and obeying the operationality criterion. Analytical learning methods are based on the Noos introspective capabilities for inspecting the episodic model built while solving the training example.

Learning with feature terms is a relational learning. Systems that also perform relational learning are ILP systems [Muggleton and De Raedt, 1994]—that use horn clauses.

Learning in Noos can be performed either on descriptions or on methods. The different learning methods incorporated to Noos perform different kinds of learning: inductive learning methods have been used for learning on descriptions and not for learning on methods. CBR methods have been used for reusing and adapting both—descriptions and methods. Finally, analytical learning methods has been used for acquiring new methods.

In Chapter 6 we will show how several learning methods have been developed and integrated to different applications built in Noos.