UNIVERSITAT POLITÈCNICA DE CATALUNYA

Departament de Llenguatges i Sistemes Informàtics

Programa de Doctorat en Intel·ligència Artificial

TESI DOCTORAL

The Noos Representation Language

Setembre 1997

Memòria presentada per en Josep Lluís Arcos i Rossell per optar al títol de Doctor en Informàtica per la Universitat Politècnica de Catalunya. El treball contingut en aquesta memòria ha estat realitzat a l'Institut d'Investigació en Intel·ligència Artificial (IIIA) del Consell Superior d'Investigacions Científiques (CSIC).

Director: Enric Plaza i Cervera

Resum

L'objectiu d'aquest treball és el disseny i implementació d'un llenguatge de representació per desenvolupar sistemes basats en coneixements que integrin resolució de problemes i aprenentatge automàtic. La nostra proposta és que aquest objectiu es pot assolir mitjançant un llenguatge de representació amb constructes de representació pròxims als entorns de modelització de coneixements, amb una memòria episòdica i amb capacitats reflexives.

Hem desenvolupat Noos, un llenguatge de representació centrat en objectes i reflexiu pròxim als entorns de modelització de coneixements. Noos es basa en el principi de descomposició tasca/mètode i en l'anàlisi dels models requerits i construïts pels mètodes de resolució de problemes. Hem formalitzat Noos utilitzant termes de característiques (feature terms), un model per formalitzar representacions centrades en objectes que ens permet integrar diverses tècniques d'aprenentatge.

La integració de l'aprenentatge automàtic en un sistema basat en coneixements implica que la modelització de coneixements del sistema ha d'incloure la modelització dels objectius de l'aprenentatge. A més, les tècniques d'aprenentatge automàtic han de poder ser modelitzades dins de l'entorm de modelització de coneixements, així com també els models requerits i construits per aquests. La integració de les tècniques d'aprenentatge automàtic requereixen de la capacitat per accedir (introspecció) a problemes ja resolts (que nosaltres anomenarem episodis) i de la capacitat de poder modificar el coneixement del sistema.

La segona proposta és que els mètodes d'aprenentatge automàtic són mètodes (en el mateix sentit que els mètodes de resolució de problemes), amb capacitats introspectives, que també poden ser analitzats com a descomposició de tasques/mètodes. Així, els mètodes d'aprenentatge automàtic poden ser representats uniformement com a mètodes i integrats en el nostre llenguatge.

La tercera proposta és que sempre que es requereix algun coneixement per un mètode de resolució de problemes, i aquest coneixement no és directament disponible, sorgeix una oportunitat per aprendre. Aquestes oportunitats les anomenem *impassos*, seguint la terminologia de SOAR, i la integració de l'aprenentatge es realitza mitjançant mètodes d'aprenentatge capaços de resoldre aquests impassos.

Aquesta memòria descriu les capacitats del llenguatge de representació Noos i el seu ús en el desenvolupament de sistemes basats en coneixements que integren resolució de problemes i aprenentatge automàtic. També presentem exemples d'aplicacions desenvolupades utilitzant Noos.

Abstract

The aim of this thesis is the design and implementation of a representation language for developing knowledge systems that integrate problem solving and learning. Our proposal is that this goal can be achieved with a representation language with representation constructs close to knowledge modeling frameworks and with episodic memory and reflective capabilities.

We have developed Noos, a reflective object-centered representation language close to knowledge modeling frameworks. Noos is based on the task/method decomposition principle and the analysis of models required and constructed by problem solving methods. Noos is formalizated using feature terms, a formal approach to object-centered representations, that provides a formalism for integrating different learning techniques.

The integration of machine learning tasks has as implication that the knowledge modeling of the implemented knowledge system has to include modeling of learning goals. Moreover, machine learning techniques have to be modeled inside the KM framework and the knowledge requirements of ML have to be addressed. The integration of learning requires the capability of accessing (introspection) to solved problems (that we call episodes) and of modifying the knowledge of the system.

The second proposal is that learning methods are methods (in the sense of knowledge modeling PSM) with introspection capabilities that can be also analyzed in the same task/method decomposition way. Thus, learning methods can be uniformly represented as methods and integrated into our framework.

The third proposal is that whenever some knowledge is required by a problem solving method, and that knowledge is not directly available, there is an opportunity for learning. We call those opportunities impasses, following SOAR terminology, and the integration of learning is realized by learning methods that are capable of solving these impasses.

In this memory we describe the capabilities of the Noos representation language and they use for developing knowledge systems that integrate problem solving and learning. Examples of applications developed in Noos will be also presented.

Contents

List of Figures

Agraïments

Aquest treball ha estat possible, principalment, gràcies al recolzament, dedicació i esforç del meu director de tesi Enric Plaza. Vull agraïr l'oportunitat que m'ha ofert per introduir-me i guiar-me en la recerca, així com també la seva paciència per anar corregint les incomptables versions dels capítols que formen aquesta memòria.

Vull també donar les gràcies a tota la gent del IIIA, tant al personal científic com al administratiu, pel seu ajut i recolzament. Part del treball reflexat en aquesta memòria és el fruit de múltiples discussions amb molts d'ells.

Voldria fer un especial esment a tots els usuaris que han gosat posar-se a desenvolupar aplicacions en Noos a la vegada que el llenguatge s'estava gestant. Especialment voldria agraïr la comprensió mostrada a en Martí Cabré, a en Jordi Sabater i, especialment, a l'Eva Armengol. L'Eva és la que més ha hagut de patir amb els canvis que hem anat introduint a Noos fins a aconseguir una versió estable del llenguatge. Vull també agraïr a en Francisco Martín el seu treball per fer accessible Noos a través d'internet.

També vull agraïr el recolzament de l'Ulises Cortés, el meu tutor i responsable de que fiqués el nas en la intel·ligència artificial.

Voldria també fer esment que aquest treball s'ha enriquit també dels suggeriments fets pels revisors anònims que han llegit el resum previ d'aquesta tesi així com les diverses publicacions internacionals presentades.

Finalment, voldria agraïr el recolzament de la família i amics i, especialment, de la Marta. Gràcies per la vostra paciència i comprensió mostrada en tot moment. Voldria fer un especial esment a la Mireia, que tot i que en el moment d'escriure aquestes ratlles encara no ha nascut, ha estat l'estímul final per poder acabar aquesta memòria.

Aquest treball ha estat finançat pels projectes de recerca AMP (CICYT 801/90 c02), ANALOG (CICYT-122/93), SMASH (TIC96-1038-C04-01), per la xarxa europea d'excel·lència ML-NET (ESPRIT 7115) i per una beca per a la recerca del Consell Superior d'Investigacions Científiques.

Als meus pares,
Lluís i Amàlia.

Chapter 1

Introduction

The main goal of this thesis is the design and implementation of a representation language for developing knowledge systems that integrate problem solving and learning.

The result has been the design and implementation of Noos, a reflective object-centered representation language for integrating inference and learning components in a uniform representation.

Our approach builds upon a variety of preceding work: the knowledge-level analysis of knowledge systems, the knowledge modeling frameworks developed for the design and construction of knowledge systems, and the research on integrated architectures for problem solving and multistrategy learning.

1.1 Motivation

One of the key issues in the current development of knowledge systems is the degree to which different components can be described, reused, and combined in a seamless way. Moreover, the current development of knowledge systems increases the necessity of incorporating learning capabilities to knowledge systems. Currently, the integration of learning components is considered as an essential topic for future design, building, and maintenance of knowledge systems.

The difficulties arisen in the development of complex knowledge systems are broadly denominated the *knowledge acquisition problem.* Adapting the approach given in [Aamodt, 1991], our view is that knowledge acquisition is a cyclic process required at the development phase and also at the problem solving phase of a knowledge system (see Figure 1.1).

The goal of the development phase is to perform a knowledge modeling analysis of the knowledge required for solving the problem and to design an application using a specific computer language. First, knowledge modeling methodologies are used to acquire knowledge models from human experts and represent these knowledge models in an implementation independent representation formalism. Specifically, the knowledge-level analysis of expert systems and the knowledge

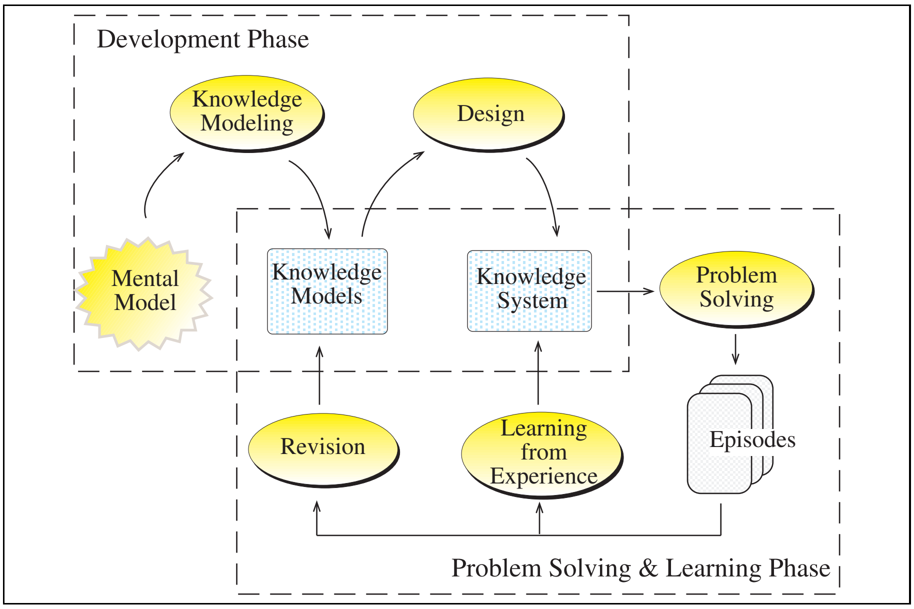


Figure 1.1. The design and maintenance cycle of knowledge systems (adapted from [Aamodt, 1991]).

modeling frameworks are methodologies developed for describing and reusing knowledge systems components. These knowledge modeling frameworks, like KADS [Akkermans et al., 1993], CommonKADS [Wielinga et al., 1993] or Components of Expertise [Steels, 1990], are based on the task/method decomposition principle and the analysis of knowledge requirements for problem solving methods (PSM).

After the knowledge modeling process (see Figure 1.1), a design process has to be performed for translating the knowledge models to a specific language constructing an executable knowledge system. A first issue arises here, since some knowledge models built by the KM methodologies are not directly operationalizable. Different languages—that partially implement the knowledge models built by KM frameworks—have been developed. Nevertheless, an active research activity is focused in developing fully operationalizable representation languages with highly expressiveness capabilities.

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. Nevertheless, learning is only used in the development phase. That is to say, the knowledge system designed is not to be capable of learning. The reason of this limited use of learning is that the working hypothesis of knowledge modeling methodologies is that the complete knowledge model of the problem can be acquired and modeled statically in the development phase.

Our proposal is that certain knowledge acquisition tasks can be *delayed* from the knowledge system development phase to the phase in which the knowledge system is actually used in the task environment. That is to say, the knowledge system can be designed with learning capabilities. Since knowledge modeling methodologies view KA as a process that basically build models, our approach means that some models are not built in the development phase—or, in general, a preliminary model is build but needs to be improved—and their construction is delayed to the problem solving phase where appropriate ML methods are appointed to generate those models.

The delay of KA tasks has as implication that the knowledge modeling of the implemented knowledge system has to include modeling of KA goals. Moreover, machine learning techniques have to be modeled inside the KM framework and the knowledge requirements of ML have to be addressed. The integration of learning requires the capability of accessing (introspection) to solved problems (that we call *episodes*) and of modifying the knowledge of the system (self-modification).

Moreover, we want to provide a framework for developing—and integrating with problem solving—several symbolic learning methods from the knowledge modeling analysis of application domains.

The contributions of machine learning have grown in the last years and have arisen many different learning techniques. Some examples of learning methods are: empirical learning or inductive learning, analytic learning or explanation based learning, memory based learning, case-based learning, learning by analogy, connectionist learning, genetic algorithms, learning by discovery, etc [Carbonell, 1989, Moreno et al., 1994].

Each different learning method is useful for specific tasks satisfying a set of requirements. For instance, analytical learning methods require a copious domain theory and few examples, while case-based learning methods require many solved cases and may also use domain knowledge. A difficulty with learning techniques is that different learning systems use different representation languages and different knowledge model about the architecture (the self-model).

There are two main approaches adopted to integrate learning with problem solving: (i) combining different learning methods in hybrid systems in order to be applied in a set of broader tasks, like CASEY [Koton, 1989] or BOLERO [López and Plaza, 1993], or (ii) developing a computational framework capable of integrating different learning methods in a uniform way. Examples of this second approach are architectures like PRODIGY [Carbonell et al., 1991] and THEO [Mitchell et al., 1991].

Integrated architectures propose different frameworks for integrating problem solving and learning. A difficulty with integrated architectures is that their representation languages are far from knowledge modeling frameworks (for instance, SOAR representation language is based on productions). Moreover, all these architectures integrate diverse learning mechanisms but present some limitations in their integration.

Our approach in developing Noos was influenced for the research on integrated architectures. Nevertheless, our approach of integration of learning techniques is restricted to symbolic learning techniques: we are focused on the integration of inductive learning techniques, case-based reasoning techniques, and analytical learning techniques. Another issue not covered by this thesis is the revision process (see Figure 1.1) for re-designing knowledge models from the experience acquired in solving problems.

One of the most research activity in machine learning, at the theoretical and practical level, is focused in the integration of several disparate learning techniques [Langley, 1989]. This research is focussed on the development of systems called *multistrategy learning systems* (MSL systems) [Michalski, 1993]. MSL systems use several learning techniques, that are combined using a specific strategy, for learning from a greater variety of inputs and acquire more flexible knowledge. Consequently, MSL systems potentially can be applied to a wide range of problems. Following this direction, our approach in Noos is to provide a representation language for facilitating the integration of several learning techniques in a knowledge system. Moreover, a domain-specific analysis for each Noos application determines different strategies for combining these several learning techniques.

1.2 Goals and contributions of the thesis

The aim of this thesis is the design and implementation of a representation language for developing knowledge systems that integrate problem solving and learning. Our proposal is that this goal can be achieved with a representation language with representation constructs close to knowledge modeling frameworks and with *episodic memory* and *reflective* capabilities.

The second proposal is that learning methods are methods (in the sense of knowledge modeling PSM) with introspection capabilities that can be also analyzed in the same task/method decomposition way. Thus, learning methods can be uniformly represented as methods and integrated into our framework.

The third proposal is that whenever some knowledge is required by a problem solving method, and that knowledge is not directly available there is an opportunity for learning. We call those opportunities *impasses*, following SOAR terminology [Newell, 1990], and the integration of learning is realized by learning methods that are capable of solving these impasses.

We have developed the Noos representation language, a language close to knowledge modeling frameworks, for developing knowledge systems that allows the integration of problem solving and learning.

Furthermore, Noos is formalizated using *feature terms*, a formal approach to object-centered representations, and providing a uniform formalism for integrating different learning techniques.

Noos is a language with capabilities of :

Representation : Noos is based on task/method decomposition principle and the analysis of knowledge requirements for methods. This capability allows

Noos to take advantage of the KA methodologies and libraries developed in KM frameworks. There are several specification languages developed in KM frameworks that can be used for knowledge modeling of applications. A knowledge modeling analysis realized in some of these specification languages can be then implemented in Noos. Nevertheless, this operationalization step is not structure-preserving with respect to knowledge modeling frameworks.

An object-centered representation language : Noos provides a structured representation of knowledge allowing the description of complex knowledge. Moreover, the Noos approach to knowledge representation provides a natural way to describe partial knowledge amenable to extension.

Introspection : Noos inference behavior is partially represented in the language itself. This reified behavior can be accessed by problem solving methods written in Noos. Introspective capabilities form the basis that allows Noos programs to reason about the system behavior. Learning methods are methods that use introspection.

Episodic memory : problems solved in Noos are automatically memorized (stored and indexed) and amenable to be accessed and reused in solving new problems. The problem solving behavior in these solved problems is also represented in Noos and, using introspection, is accessible and inspectable. Episodic memory can be inspected by means of three different access mechanisms: *access by path*, that provides a way to access to specific portions of the episodic memory; *retrieval methods*, that provide a mechanism for content-based access to the episodic memory; and *perspectives*, a mechanism to describe declarative biases for case retrieval in the structured representation of cases. The memorization of solved problems is a basic building block—together with introspection—for integrating learning into our KM framework.

Integrated Problem solving and learning : the integration of learning in Noos is based on two proposals:

- Learning methods are methods (in the sense of knowledge modeling PSM) that can be also analyzed in the same task/method decomposition way. Introspective requirements of learning methods can be fulfilled using Noos reflective operations for accessing the episodic memory.
- *Impasse driven learning*: whenever some knowledge is required by a problem solving method, and that knowledge is not directly available, there is an opportunity for learning. We call those opportunities *impasses*, following SOAR terminology, and the integration of learning is realized by learning methods that are capable of solving these impasses.

Machine learning methods : Noos provides a collection of basic mechanisms allowing the development of different learning methods such as inductive learning methods, CBR, and analytical learning methods.

- *Inductive learning methods* are developed in Noos as search methods (that follow certain biases) over the space of feature terms. Domain specific knowledge is used for constructing inductive learning methods that follow different searching biases. Inductive learning methods are based on the feature term subsumption and antiunification operations of Noos.
- *Case-based reasoning methods* are developed in Noos as problem solving methods with lazy learning capabilities that search for previously similar solved problems in the Noos episodic memory. CBR methods are based on the retrieval and subsumption operations of Noos.
- *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 in Noos while solving the training example.

A multistrategy learning approach : different learning methods can be integrated in Noos using a common scheme based on three main subtasks: *Introspection*, *Construction*, and *Revision*. Using this common scheme, we will present how different learning methods can be designed and integrated in a problem solving system.

Noos has been implemented using Common Lisp and currently is running in several platforms. A window-based graphical interface has been also develpped in Macintosh Common Lisp [Digitool, 1996] for the MacOS version of Noos. Although the implementation details are not the main focus of this thesis, we provide a brief description of the Noos development environment in Appendix A.

Moreover, Noos has been used by several persons to develop several applications that integrate different problem solving methods and different learning methods. The research done in one of them, called *Saxex* and developed for myself, has been awarded with the "Swets & Zeitlinger Distinguished Paper Award" at the 1997 International Computer Music Conference.

1.3 Structure of the thesis

This thesis is organized in seven chapters, including this introduction chapter, and four Appendices:

Chapter 2 reviews the main research relevant to our thesis and discuss their contributions and limitations. We present the research on knowledge modeling methodologies for analyzing and developing knowledge systems; the research on integrated architectures for exploring the relationships among problem solving, learning, and knowledge representation; the research on reflective representation languages for providing introspection capabilities on knowledge systems; and finally, discuss the the role of reflection on learning.

Chapter 3 presents the Noos representation language. The language is introduced incrementally. First, The Noos modeling framework is presented. Next, the basic elements of the language such as descriptions, refinement, references, methods, and the basic inference are described. Then, the reflective capabilities of Noos are described introducing elements such as metalevels, tasks, reflective operations, reification, and reinstantiation. Next, a declarative mechanism for decision making about sets of alternatives, called *preferences*, is presented. Finally, the complete Noos inference engine is described.

Chapter 4 presents the Noos capabilities for reasoning about experience and the integration of learning and problem solving. First, the notion of *episodic memory* is introduced. Then, introspective mechanisms such as *retrieval* and *perspectives* are presented. Next, three different families of learning techniques such as case-based reasoning, inductive learning, and analytical learning, are presented with examples of how have been integrated in Noos.

Chapter 5 presents feature terms, a formalism for describing the Noos language. Feature terms are introduced using a syntax notation based on the λN calculus. Then, using the work on feature structures a semantics based on the notion of partial descriptions is presented. The results obtained by the research on feature structures are also adapted for providing several equivalent representations of feature terms.

Chapter 5 introduces a formalism for describing preferences based on the notion of pre-orders. Two kinds of basic operations are defined over preferences: *preference operations*, that take a set of source elements and an ordering criterion and build a preference (a partially ordered set), and *preference combination operations*, that take two preferences and a combination criterion and build a new preference.

Next, using feature terms *perspectives* are defined. Perspectives are formalized as second order feature terms that denote sets of terms.

Finally, we describe formally the inference in Noos using Descriptive Dynamic Logic, a propositional dynamic logic for describing and comparing reflective knowledge systems.

Chapter 6 provides a set of examples of how diverse applications have been developed using Noos by several persons at the IIIA. Specifically, the chapter present six applications developed using Noos: CHROMA, SPIN, SHAM, GYMEL, *Saxex*, and NoosWeb.

Chapter 7 summarizes the main contributions of the thesis and discuss further directions of research.

Appendix A presents the Noos development environment including development facilities such as browsing, tracing, and the extension of the built-in methods.

Appendix B provides a glossary of the main concepts introduced in this thesis.

Appendix C presents the complete syntax of the Noos language, using BNF notation, and the collection of compact descriptions for built-in methods.

Appendix D presents the complete list of Noos built-in methods describing their required features and their functionality.

Chapter 2

Background

The goal of this chapter is to describe the literature relevant to our work. On the one hand, in Section 2.1 we will present the research on knowledge modeling methodologies for analyzing and developing knowledge systems. Then, in Section 2.2 we will present the research on integrated architectures for exploring the relationships among problem solving, learning, and knowledge representation. In Section 2.3 we will focus on the research on reflective representation languages for providing introspection capabilities on knowledge systems. Next, in Section 2.4 we discuss the role of reflection and learning. Finally, in Section 2.5 we provide some conclusions about these research approaches.

2.1 Knowledge modeling frameworks

Knowledge modeling frameworks (KMF) propose methodologies for analyzing and developing knowledge systems. Different approaches differ on the methodology they propose but all of them are based on the conception of constructing a *conceptual model* of a system which describes the required knowledge and inferences at an implementation independent way.

Knowledge modeling frameworks are highly influenced by previous work on *Knowledge Level* [Newell, 1982], *Generic Tasks* [Chandrasekaran, 1986], and Problem Solving Methods [McDermott, 1988].

Knowledge Level proposes a level of description of systems focused in describing the knowledge they contain rather than the implementation structures of these knowledge. A system at knowledge level is viewed as an agent with three components: goals, actions, and bodies. This description level provides a more adequate level of description of knowledge systems. A *principle of rationality* rule guides the behavior of the agent: Actions are selected to attain its goals.

Generic Tasks proposes a task-oriented methodology for developing knowledge systems. This methodology focuses the process of analyzing and building a knowledge system for given problem by representing a *task-structure* for the problem and specifying the domain requirements of the tasks in the task structure. A task structure is described in terms of the methods that are applicable to tasks and the conditions under which each method is applicable. Each method is itself specified in terms of how it uses knowledge and inference to achieve a task, and in terms of which subtasks are required to be achieved before it can succeed. This decomposition is done recursively until methods which achieve tasks are not decomposed by additional subtasks. The task structure offers the advantage that directs the knowledge acquisition, since knowledge and inference requirements for the methods can be explicitly identified. The generic tasks methodology is also based on the view that the task analysis is aided by the fact that a number of generic tasks and methods have been identified. This library of generic components facilitates the development of knowledge systems.

Problem Solving Methods (PSMs) are knowledge use characterizations of how problems can be solved. Each problem solving method is described using a set of roles that have to be filled by domain models. The advantages of focusing the development of knowledge systems on the problem solving methods is that the roles required by PSMs prescribe what domain knowledge has to be acquired. A current research work on PSMs is focused on *assumptions* over domain models required to perform the PSM. Assumptions facilitate the application of PSMs in tasks by assuming they use a common terminological structure (see for example the work on assumptions for model-based diagnosis in [Fensel and Benjamins, 1996]). The research on PSMs is also focused on *reuse*. The progress on reuse of PSMs is an important aspect that will reduce considerably the development efforts of knowledge systems.

Below we will describe briefly two methodologies proposed for the analysis and development of knowledge systems: The *Common*KADS methodology and the Components of Expertise methodology. Next, we will present Krest, a knowledge-system design tool implementing the kernel of the componential methodology. Finally, we will present another knowledge engineering environment, called Protégé-II, which supports the construction of knowledge systems from reusable components.

CommonKADS

*Common*KADS [Wielinga et al., 1993] [Van de Velde, 1994a] is a methodology for the development of knowledge systems. *Common*KADS is the evolution of the KADS methodology [Wielinga et al., 1992] [Schreiber et al., 1993].

In *Common*KADS the development of a knowledge system is viewed as the construction of models of problem solving behavior in a concrete organizational context. *Common*KADS provides a set of models, called the *model set*, that allow for expressing the various perspectives of the problem solving situation: *organization model*, *task model*, *expertise model*, *communication model*, *agent model*, and *design model*.

The *organization model* describes the organizational context in which the knowledge system to be developed occurs (e.g. resources, functions, and processes).

The *task model* describes the tasks and activities that are performed for realizing organizational functions (e.g. tasks, task inputs, and required capabilities).

The *agent model* collects relevant properties of the different agents involved in the realization of tasks described in the task model (e.g. users and software systems).

The *communication model* describes communication processes among agents (described in terms such as transaction plans, ingredients, and initiatives).

The *expertise model* describes the knowledge of an agent relevant to solve a specific task and its use-specific structure. This knowledge and structure of the expertise model is described as three kinds of knowledge models (*domain knowledge*, *task knowledge*, and *inference knowledge*) and with a set of mappings among them.

The *design model* describes the realization of problem solving behaviors described in expertise and communication model in computational and representational terms.

Organization model, task model, agent model, and communication model capture the context in which the problem solving activity is performed. The expertise model captures the knowledge and reasoning involved in performing tasks. The design model describes the computational realization.

Here, due to the relevance with our work, we will describe in more detail the expertise model. The purpose of the expertise model is to describe the collection of elements of knowledge involved and their roles in solving a specific task. In *Common*KADS the different roles are captured in three basic knowledge categories: *task knowledge*, *domain knowledge*, and *inference knowledge*.

The *task knowledge* category describes a recursive decomposition of a top-level task in (sub)tasks, specifies what it means to achieve these tasks, and describes when these (sub)tasks are to be executed in order to achieve the parent task (describes the control). A task description consists of two parts: a task definition and a task body. The task definition specifies what it means to achieve the task by defining its goal in terms of input and output roles. The input and output roles are references to parts of domain knowledge. The task body describes how the task can be achieved by means of describing a set of subtasks, a set of additional roles, and a control structure.

A usual view of the task knowledge is the *task decomposition* of a top level task. The task decomposition shows the different (sub)tasks that contribute to the achievement of parent tasks and, at the leaves of the structure, the set of inferences (from inference knowledge category) that can accomplish primitive tasks.

The *domain knowledge* category specifies the form, structure, and contents of domain specific knowledge that is relevant for an application. The form and structure of domain specific knowledge is defined specifying different *ontologies*, that provide partial coherent views on parts of the domain knowledge. A collection of domain knowledge statements described by a particular ontology, and describing a specific problem context, is called a *domain model*. Domain knowledge is structured in a series of domain models.

The *inference knowledge* category specifies the primitive reasoning steps (*inferences*) in an application. Inference knowledge also describes the *knowledge roles* that refer to classes of domain knowledge elements manipulated by the inferences.

The inferences modeled in a particular application form a structure called the *inference structure* that describes the data dependencies between the different inferences. An inference structure, however, does not express a control diagram. It only expresses how knowledge elements referred to by the various roles are used and produced by the various inferences.

Finally, one of the goals of the *Common*KADS project is the *reuse* of knowledge components of the model set. Libraries of reusable elements potentially available for all aspects modeled in an application have been developed.

ComMet

The componential methodology [Steels, 1990], called ComMet, proposes a framework for describing expertise at the knowledge level. ComMet decomposes the expertise into three different perspectives: the *task perspective*, the *model perspective*, and the *method perspective*.

The *task perspective* specifies the set of tasks that a problem solver has to achieve. Usually, there is a main task that covers the whole application (e.g. diagnose of car malfunctions). This task usually decomposes into several subtasks. These tasks can still be further decomposed until a non-decomposed elementary tasks.

The decomposition of tasks into subtasks is called the *task structure*. The task structure is not necessarily static. Sometimes not all subtasks are executed for each case. The task structure is represented as an and/or tree.

Problem solving in ComMet is viewed as a modeling activity. Solving a problem is viewed as the construction of a model of various aspects of the world that are relevant to find a solution to the problem.

The *model perspective* focuses on the question what knowledge is available to achieve the tasks. ComMet makes a distinction between three different kinds of models: *case models*, *domain models*, and *problem solving process models*. An important effort of knowledge analysis is focused on the development of models required in an application and the dependencies given between the different models.

Case models describes specific situations to be solved. Case models can be described using different perspectives. A *components model* is a model of a system in terms of its components and subcomponents. A *descriptive model* is a model of a system in terms of a set of observable features of the system. *Causal models*, *behavioral models*, *functional models*, and *temporal models* are examples of other case models. Usually the kind of model chose is strongly influenced by the kind of tasks to be solved. For instance, *configuration* tasks are described in terms of how the different components of a system are connected.

When the different case models have been identified, the following step is to determine how these models are related to each other. The result of this analysis is represented in ComMet as a case model dependency diagram.

Domain models are models valid for a variety of cases. Domain models describe domain-specific knowledge which is used by problem solving methods to construct case models. The domain models are added to the case model dependency diagram to yield a complete model dependency diagram. The domain models can be divided into two classes: the *expansion models*, containing knowledge relevant for expanding a model (e.g. for adding more symptoms to a symptom case model), and *mapping models*, used to construct or modify case models based on a mapping from elements of other models (e.g. from symptoms to malfunctions).

Problem solving process models are needed when the problem solver has to reason about itself.

The *method perspective* specifies how and when the knowledge is applied. A method is a code that imposes a control structure between tasks and is represented in a *control diagram*. A control diagram is a graph where the nodes correspond to subtasks and transitions between nodes represent the control flow between different tasks. Methods are divided into three main classes: *task decomposition methods*, *task execution methods*, and *search methods*.

Task decomposition methods decompose a task into subtasks and put a control structure on the subtasks. The subtasks of task decomposition methods still need to be further decomposed.

Task execution methods also decompose a task into subtasks and put a control structure on the subtasks. The subtasks of execution methods are implementation tasks whose execution results in problem solving activities.

Search methods are necessary when not enough information is available to decide in which way a case model has to be developed. Search methods are used to choose, among several alternatives, the most appropriate.

KresT

Krest [Jonckers et al., 1992] is a knowledge-system design tool implementing the kernel of the componential methodology. According to this methodology, the basic components of KresT are tasks, methods, and models. These components are described by a knowledge-level description language provided by a software extension called *application kit* (or *appkit*). The description language is based of feature structures.

Components can be connected together with different types of relations. According to ComMet, any component can be linked to any other, but KresT implements *task-roles*, which link tasks to models or other tasks, and *method roles*, which link methods to models or tasks.

Task roles relate tasks to models using two roles: *input roles*, relating the models required by a task, and *output roles*, relating the output models of a task. A task can be related to other tasks using a *subtask role*.

In KresT each task is connected to exactly one method and each method may perform only one task. The rationale for that constraint comes from the fact that the methodology is case oriented: methods are viewed as instances of algorithms, applied in a particular context, and with particular input and output roles.

Method roles relate methods to models and tasks. When a task role is established between a task and its method, a method role is also automatically created for the method with the task. Method roles hold a set of properties. The four basic properties of a method role are: its *name*, used as a key to select it and to represent it on the diagrams, its *type*, inherited from the task role, the constraints imposed to candidate fillers (related to input and output models), and the *multiplicity* of the role, whether it allows multiple fillers.

Components are grouped in a larger units, called *projects*. A project is a set of related tasks, models, and methods required to build a particular system.

The interface of KresT allows to design components by editing feature structures and through a graphical interface that provides a set of diagram types such as a *task-structure-diagram* representing the task/subtask hierarchy, a *task-model-dependency-diagram* representing task/model relations for one task, together with information about the method/model relation, and a *subtask-model-dependency-diagram* representing in a same window the same relations for all subtasks of a given task.

Protégé-II

The Protégé-II [Puerta et al., 1992] is a knowledge engineering environment that allows developers to build knowledge systems by selecting and modifying reusable problem solving methods and domain ontologies. Protégé-II provides a suite of knowledge-acquisition tools to generate domain-specific knowledge-acquisition tools from ontologies. A main goal of Protégé-II is to support early prototyping.

The model of reuse in Protégé-II is based on the notion of a library of problem solving methods that performs tasks. PSMs have input-output requirements and are decomposable into subtasks. Other methods can perform these subtasks. Methods that are not further decomposed are called *mechanisms*.

Ontologies in Protégé-II are defined as class hierarchies. There are three main types of ontologies in Protégé-II: *domain ontologies*, *method ontologies*, and *application ontologies*. Domain ontologies model concepts and relationships for a particular domain. Method ontologies model concepts related to problem solving methods, including input and output requirements. To enable reuse, method ontologies should be domain independent. Application ontologies combine domain and method ontologies for configuring a particular application.

Since a main goal of Protégé-II is to provide support for reuse of problem solving methods and ontologies, it also provides a way to connect these two components in an application called *mapping relations* [Gennari et al., 1994]. Mapping relations encode any adaptations from domain models to problem solving methods.

Protégé-II provides a *mappings ontology* for guiding the mapping process. Mappings ontology defines a set of mapping relations (e.g. *renaming mappings* applied when the semantics between method and application classes match but slot name have to be translated) that constitute the language for specifying mapping relations.

2.2 Integrated architectures

The goal of the research on integrated architectures is to explore the relationships among problem solving, learning, and representation.

Different integrated architectures have divergent features that lead to different *properties* concerning to the problem solving process (e.g. forward and backward chaining, or impasse-driven inference), the architecture organization (e.g. hierarchical or modular), the knowledge representation language and knowledge structures used (e.g. uniform representation or heterogeneous representation), and the learning capabilities (e.g. deliberative/reflexive, monotonic/nonmonotonic).

These design decisions then lead to the support of specific *capabilities* such as capabilities related to problem solving (e.g. planning, self-reflection, or metareasoning) or capabilities related to learning (e.g. single or multiple learning methods, inductive learning, explanation-based learning, or learning by analogy).

The choice of features is often made by following some explicit *methodological assumptions*, often driven by the domains and environments in which the architecture will be used.

A detailed comparison study of twelve different integrated architectures can be found in [Wray et al., 1995]. In this monograph we will describe three architectures that were influential in the initial design and implementation of Noos: THEO, SOAR, and PRODIGY. We will focus the description of such systems in four aspects: the methodological assumptions, the representation language, the problem solving process, and the integration of learning mechanisms.

THEO

The design of the THEO architecture [Mitchell et al., 1991] was motivated by the goal of providing a framework to support basic research on general problem solving, learning, and knowledge representation, and especially on the interactions among these three issues. A second motivation was to design an efficient framework for developing effective knowledge systems. THEO is intended to provide several levels of sophistication and efficiency depending on the user's requirements.

THEO utilizes a uniform frame-based knowledge representation. Frames represent entities and have a collection of slots representing relations among entities. The integration of a frame with a slot and a value is said to be a belief. Specifically, an assertion of the form `(entity slot) = value` represents the belief that some entity named `entity` stands in some relation named `slot` with another entity named `value`. For example, we might assert `(fred wife) = wilma`.

Slots are themselves entities. A number of slots for describing slots are predefined in THEO. Predefined slots of slots describe different types of information about slots. Some of these slots of slots describe various slot properties such as `domain`, `range`, `inverse`, and `transitive?`. Other slots of slots describe how to infer values of instances of the slot, such as `toget`, `methods`, and `available.methods`. Slots such as `whentocache` and `whentoebg` describe control information determining whether once the value of the slot is inferred it has to be cached. In addition slots of slots such as `explanation` and `dependents` describe information about the interdependencies among slot values.

Slot values in THEO can be inferred using methods. There are three layers of specification of methods for a particular slot. At the most basic layer, THEO allows to associate a Lisp function to a slot by asserting a value for the `toget` slot of that slot. At a second layer, the user can specify a list containing some of system predefined inference methods (such as `inherits`, `default.value`, and `drop.context`) in the `available.methods` slot of the slot. Finally, the higher layer allows to associate the `defines` method. The `defines` method specify the use of definitions for other slots or the use of methods built by THEO's explanation-based learning mechanism.

Problem instances in THEO are pairs of the form `(entity slot)`, representing the task of determining a justifiable belief of the form `(entity slot) = value`. For example, we might pose the problem `(fred wife)` whose solution is `wilma`.

Problem classes, or sets of problem instances, are described either by a single token `slot` (e.g. `wife`) representing the class of problem instances of the domain of `slot`, or by a pair of the form `(entity slot)` (e.g. `(male wife)`) representing the class of problem instances of `slot` and whose entity is a member of the class represented by `entity`.

Problem instances and problem classes are themselves entities. Thus, THEO can hold beliefs about problems and pose problems regarding problems just as it can for any other entity. For example, we could assert the belief that the problem of determining Fred's wife is difficult by asserting `((fred wife) difficult?) = true`.

Problem solving in THEO corresponds to inferring the value of a slot. Inference in THEO is *impasse-driven*: when a problem instance is posed to the system and whose slot value is unknown, an impasse arises, resulting in the subtask of inferring the corresponding slot value.

Given a problem instance `(E S)`, the THEO's inference mechanism is based on three inference layers:

- *Layer 1*: apply the Lisp function specified in slot `toget` of slot `S` of entity `E`. If unspecified, go to level 2.

- *Layer 2*: apply the list of methods specified in slot `available.methods` of slot `S` of entity `E` until a value is obtained. If unspecified, the default value of the `available.methods` slot is the list `(defines inherits drop.context default.value)`. The `defines` method starts layer 3.

- *Layer 3*: Utilize definitions for other slots inferred from knowledge of slots of slot such as `inverse` and `transitive`, or definitions inferred by THEO explanation-based learning mechanism.

Once a problem is solved, THEO may store the solution in the memory, asserting the corresponding belief, indexed by the problem name—or in other words, by the problem class. When it stores such beliefs, it also stores the *explanation* justifying the new assertion in terms of the beliefs on which depends.

One of THEO's guiding principles was that all knowledge should be open for inspection. Because of the uniform frame-based representation of THEO, the access to any knowledge in THEO is performed by accessing to the value of corresponding slot. If desired knowledge is not present, an impasse occurs and the THEO inference mechanism is automatically engaged. This knowledge transparency along with the automatic storage of solved problems is the basis for the incorporation of learning mechanisms in THEO.

Three learning mechanisms are available in THEO: caching of inferred values, explanation-based learning of macro-methods for inferring slot values, and inductive learning for ordering the methods for inferring slot values (methods in `available.methods` slot).

Caching is the simplest form of learning in THEO. Once the value of a slot is inferred, the value may be stored in the slot, along with an explanation of how the value was inferred. This mechanism decreases the cost of slot accesses in future demands.

Explanations of how slot values are inferred constitute the input knowledge for explanation-based learning. In particular, after THEO infers the value of some slot `S`, it forms a macro-method, using a module called TMAC, by examining explanations of previously successful slot inferences. This macro-method is stored in a slot of the slot `S'` from which `S` was specializated. Then, THEO will be able to use this macro-method when attempting to infer values for other slots that are specializations of `S'`.

Since THEO usually has several methods available for inferring the value of any given slot, THEO also improves its performance modifying the order in which slot inference methods are attempted. By default, THEO attempts methods in the order in which they are listed in the `available.methods` slot. In order to learn a better ordering for methods, THEO keeps a set of statistics on the cost and likelihood of success for each slot inference method. THEO incorporates an inductive learning method called SE [Etzioni, 1988] for ordering methods. SE accepts as input a slot address and the corresponding list of available methods, and produces as output a list of methods ordered in the sequence they should be attempted.

SOAR

Research on SOAR [Laird et al., 1987] [Rosenbloom et al., 1991] is focussed on the development and application of an architecture for general intelligence. SOAR is based on formulating all goal-oriented behavior as search in problem spaces. A problem space determines a set of states and operators that can be used during the processing to attain a goal. Each goal defines a problem solving context that contains, in addition to a goal, the roles for a problem space, a state, and an operator.

All knowledge in SOAR is stored in two memories: the short-term memory (also called working memory) and the long-term memory.

Working memory consists of a set of objects and preferences about objects. Each object in the working memory has a class name, a unique identifier, and a set of attributes with associated values, which may be constants (e.g. numbers) or identifiers. For example a particular person could be represented by the following object:

```
(person Susan ^profession engineer ^age 33)
```

Preferences are architecturally interpretable elements that describe the acceptability, desirability, and necessity of selecting particular problem spaces, states, and operators. The context in which a preference is applicable is specified by its goal, problem-space, state, and operator attributes. There are two types of acceptability preferences (acceptable and reject) to select an operator, five types of desirability preferences (worst, worse, indifferent, better, and best) to determine the desirability of objects, and two necessity preferences (require and prohibit) to select and object for achieving a goal. For example, the following is a desirability preference stating that operator `o1` is the best operator for state `s1`, problem space `p1` and goal `g1`:

```
(preference p1 ^role operator ^value best ^goal g1
        ^problem-space p1 ^state s1)
```

All long-term knowledge is stored in form of productions. Productions have a set of conditions, which are patterns to be matched to working memory, and a set of actions to perform when the production fires.

Problem solving in SOAR is decomposed in two phases: the elaboration phase and the decision phase. This two phases are repeated until the goal of the current task is reached. During the elaboration phase all productions that match the current working memory are fired in parallel, and this is repeated until no more productions are matched. The elaboration process retrieves into working memory new objects, new information about existing objects and new preferences. The decision phase examines any preferences (added either in this phase, or in previous ones), and chooses the next problem space, state, operator or goal to place in the context stack. The decision phase may change any current slot values, or any previous slot values in the context stack.

If there is not enough information (or the information is contradictory) for the decision phase to proceed, then an impasse arises. There are four types of impasses: *tie* impasse, arising when two or more elements have equal preference, *no-change* impasse, arising when no preferences are in the working memory, *reject* impasse, arising when only preferences in working memory are rejected by other preferences, and *conflict* impasse, arising when two or more objects are the best choices. When an impasse occurs, a subgoal with an associated problem solving context is automatically generated for the task of resolving the impasse.

Productions into the SOAR's long term memory cannot be directly accessed by other SOAR rules. Knowledge in long term memory is only indirectly accessed by pattern matching in the elaboration phase.

All learning occurs in SOAR through the *chunking* mechanism. Chunking is a form of explanation based learning that has inductive properties. Whenever the decision cycle returns a result to a supergoal, a new production is created whose conditions are the elements tested which existed before the impasse, and whose actions are the preferences returned. This new production is called a *chunk*. The conditions of a chunk are determined by a dependency analysis. The new chunk is placed in long-term memory immediately, and is available on the next elaboration phase, thus SOAR's learning is intermixed with its problem solving.

Chunking is the only architecturally supported learning mechanism of SOAR. However, other learning mechanisms such as learning by abstraction and basic analogy have been developed from chunking and other architectural components.

Prodigy

PRODIGY [Carbonell et al., 1991] [Carbonell et al., 1995] is a general planner/problem solver that integrates multiple learning modules.

Knowledge in PRODIGY is described using a uniform representation, which is called the *Prodigy Description Language* (PDL), based on first order predicate logic. PDL includes four types of knowledge concepts:

- Ground atomic formulas: used to describe states and goals,

- Operators: used to describe what actions effect which changes to the states. Operators map states into new states. PRODIGY operators consist of a FOL expression describing the operator's preconditions, coupled with conditional *add* and *delete* lists representing the resulting changes to the state when the operator is applied. Operators may also contain conditional effects that represent changes to the world that are dependent on the state in which the operator is applied.

- Inference rules: used to deduce added information about a state. Inference rules are represented as simplified operators without *delete* lists.

- Control rules: used to guide the search. Control rules map a set of candidate decisions (such as which legal operator to apply or which goal to work on next) into a smaller or priorized decision set. Each control rule has a left-hand side condition testing applicability and a right-hand side indicating whether to *select*, *reject*, or *prefer* a particular candidate.

All the knowledge used or learned in any PRODIGY module is open for inspection and use for every other module. The uniform logic-based representation of both control knowledge and domain knowledge provides a uniform access to all knowledge.

A problem consists of an initial state and a goal expression. Solving a problem in PRODIGY—given an initial state, a goal, a set of operators, and a set of control rules—is to find a sequence of operators that, if applied to the initial state, produces a final state satisfying the goal expression. The search tree initially starts out as a single node containing the initial state and a goal expression. The tree is expanded by repeating the following two steps:

1. *Decision phase:* There are four types of decisions that PRODIGY makes during problem solving. First, it must decide what node in the search tree to expand next, defaulting to a depth-first expansion. Each node consists of a set of goals and a state describing the world. After a node has been selected, one of the node's goals must be selected, and then an operator relevant to this goal must be chosen. Finally, a set of bindings for the parameters of that operator must be decided upon.

2. *Expansion phase*: If the instantiated operator's preconditions are satisfied, the operator is applied. Otherwise, PRODIGY subgoals on the unmatched preconditions. In either case, a new node is created.

When PRODIGY is solving a problem, it makes decisions about which node to expand, which operator to apply, which objects to use, and which goal to go on. These decisions are influenced by control rules in order to increase the efficiency of problem solving search and to improve the quality of the solutions that are found. Using backtracking, the candidates are attempted, according to the preference order inferred by control rules, until all candidates are exhausted or a global solution is found.

Learning in PRODIGY is a deliberative metareasoning process: learning modules are activated when the system believes that the acquisition of new knowledge can be useful. PRODIGY has six learning modules: APPRENTICE, EBL, STATIC, ANALOGY, ALPINE, and EXPERIMENT.

APPRENTICE is a graphic-based interface for knowledge acquisition of domain knowledge and for guiding the problem solving search.

EBL is an explanation-based learning module for acquiring control rules from a problem solving trace. After a search problem solving episode, explanations are generated in a three stage process. First, EBL considers what to learn from a problem solving trace, then it considers how to best represent the learned information as control rules, and finally it empirically tests the utility of the resulting control rules to insure that they are indeed useful. Because EBL has access to the complete trace, it is used to learn from successes as well as to learn from failures.

STATIC is a learning module for learning control rules by analyzing domain descriptions. STATIC can be viewed as a compiler for PRODIGY's domains.

ANALOGY is a case-based learning module based on derivational analogy. The focus of ANALOGY is to use similar previously solved problems to solve new problems [Veloso, 1992] [Veloso and Carbonell, 1993]. The problem solver records justifications for each decision taken during its search process. These justifications are then used to guide the reconstruction of the solution of new situations where equivalent justifications hold true.

ALPINE is an abstraction learning and planning module. ALPINE is used for performing hierarchical planning in PRODIGY and is based on an analysis of the domain for automatically generating multiple abstraction levels.

Finally, EXPERIMENT is a learning by experimentation module for refining domain knowledge that is incompletely specified.

2.3 Reflective representation languages

The research on reflective representation languages is motivated by the fact that the implementation of complex knowledge systems requires the incorporation of mechanisms for reasoning about themselves. The goal is therefore the design of reflective languages for developing knowledge systems able to describe and modify themselves. Examples of reflective representation languages based on procedural reflection are RLL-1 [Greiner and Lenat, 1980], MOP [Kiczales et al., 1991], and 3-Lisp [Smith, 1985].

Moreover, the development of knowledge modeling frameworks carried out the design of formal and executable specification languages for describing and implementing knowledge systems based on these knowledge modeling frameworks (see [Fensel, 1995a]). Examples of languages are MODEL-K [Karbach et al., 1991], $(ML)^2$ [van Harmelen and Balder, 1992], and KARL [Angele et al., 1994][Fensel, 1995b] based on KADS model, or KresT [Jonckers et al., 1992] and MetaKit [Slodzian, 1994b] based on the componential framework.

RLL-1

RLL-1 [Greiner and Lenat, 1980] is a reflective representation language which was used as the basis for implementing the EURISKO system [Lenat, 1983], a system for learning by discovery. RLL-1 is a highly self-descriptive representation language developed by means of a uniform representation of frames and slots, and by possessing declarative descriptions of parts of itself.

Every component of the RLL-1 language (e.g. individual slots, modes of inheritance, and even data-accessing functions) is visible: can be represented within the language, inspected and modified. The description of all components is given also in terms of frames and slots.

KRS

KRS [van Marcke, 1987] [van Marcke, 1988] is a representation language for supporting knowledge based programming. KRS design was influenced by RLLs such as RLL-1 and ARLO [Haase, 1987]. It is a representation language defined on top of Lisp.

The primitive for representation in KRS is a large network of concepts, called the *concept-graph*. Nodes of the network are called *concepts*. Links between nodes are called *subjects*. Each concept represents a particular entity in the representation domain.

The concept graph is defined and inspected by descriptions defined in a *concept-language*. The concept-language has two important features: descriptions are interpreted such that the concept-graph is constructed in a lazy way, and the concept-language is lexically scoped.

Lazy interpretation allows to define large libraries without consuming all the memory, and allows an easy description of circular parts of the concept-graph.

Lexical scope gives all concepts described by the same description in the concept-language easy access to each other.

Inference in KRS is based on two mechanisms: *referent computation* and *inheritance*. Referent computation is a mechanism to compute concept's referent from its definition. Inheritance in KRS is the process of augmenting a concept's description by copying the subjects of the concept's type into the concept. A concept's type is also a concept. Inheritance in KRS is a single inheritance mechanism.

The Reflect Project

The aim of the REFLECT project was the development of an architecture for the construction of knowledge systems providing facilities such as knowledge-base maintenance, validation, and adaptative interaction. The claim of the REFLECT project is that more advanced knowledge systems can only be realized by a reflective reasoning architecture.

The REFLECT approach is based on the research on basic mechanisms of self-representation, causal connections, and integrated reflective computation in the same direction as FOL [Weyhrauch, 1980]. The aim of the REFLECT project is to describe an architecture by way of a generic module structure and corresponding relationships among them.

The REFLECT architecture is based on the KADS expertise model of problem solving. Following the KADS approach, it is proposed an architecture based on three layers: a *domain layer* for defining domain knowledge, an *inference layer* for defining inference knowledge, and a *task layer* for defining task knowledge.

The approach in REFLECT project was that this reflective architecture can be represented in a formal language. For this purpose, languages such as MODEL-K [Karbach et al., 1991] and $(\mathrm{ML})^2$ [van Harmelen and Balder, 1992] were developed. Below we will describe the REFLECT approach by means of the description of the $(\mathrm{ML})^2$ language.

(ML)^2

The (ML)^2 language [van Harmelen and Balder, 1992] is a language developed in the REFLECT project that provides a formal specification language for the KADS methodology. (ML)^2 combines three types of logic: extended order-sorted first-order logic for specifying the domain layer, first-order meta-logic for specifying the inference layer, and quantified dynamic logic for specifying the task layer.

Domain knowledge is modeled in (ML)^2 using an order-sorted first-order logic extended by modularization. The specification of domain knowledge can be divided into several modules. Such a module defines a signature (i.e. sorts, constants, functions and predicates) and defines axioms (i.e. logical formulae). Modules can be combined by a union operator. Constants model instances of the domain. Sorts model a class hierarchy for constants. Predicates model relationships among concepts. Functions model attributes of concepts.

Inference knowledge is modeled in (ML)^2 using a first-order meta-logic. Every inference action is described by a predicate and a logical theory (e.g. a set of input and output roles, a signature, and a set of axioms). Inferences are modeled using a meta-language of the domain knowledge that allows to express properties of relations over domain knowledge formulae. The domain layer and the inference layer are causally connected by a set of reflection rules.

Task knowledge is modeled in (ML)^2 using a quantified dynamic logic. Task knowledge models dynamic control between inference actions. Every predicate specifying an inference action in the inference knowledge is regarded as an elementary program statement and the knowledge roles are used as input and output parameters of such programs. For every inference a history variable is defined which stores the input-output pairs for every execution step. Four types of operations are available for each inference action in the task layer: checking whether an instantiation exists, checking whether an instantiation has already been computed, checking whether more instantiations exists, and actually computing and storing a new instantiation. Moreover, (ML)^2 provides a set of elementary combination operations such as sequential composition, non-deterministic iteration, and non-deterministic choice.

MetaKit

MetaKit [Slodzian, 1994b] is an extension of KresT's Basekit which provides an ontology for metalevel design. MetaKit provides a set of methods and forms to design *meta-projects* and a library of knowledge elements that can be reused in any meta-project.

A *meta-project* is a project performing operations at the meta-level of another project, called the *object-project*. Meta-projects are projects that operate on another project, using methods and content forms defined in the MetaKit. The same meta-project can be successively or simultaneously working at a meta-level above several object-projects. A meta-project may equally operate on another meta-project and there are no limitations imposed on the number of levels.

Domain models of a meta-project are models of parts of the object-project.

MetaKit provides a set of attributes and values for describing such models. For instance, the type of an object-level object is characterized by the value of the `referent` attribute, which can be `component` (task, model, or method) or `project`.

MetaKit provides also a set of metalevel methods for operating on models of methods, such as `get-method-type` and `get-method-task`, methods for working with models of models, such as `get-tasks-reading-model`, methods for operating on models of tasks, such as `get-task-input-models` and `get-task-output-roles`, methods for operating on roles, such as `fills-roles` and `get-method-role`, methods for operate on attribute representation, such as `get-attribute-value` and `unify-fragments`, methods for operating on models of projects and fragments, such as `extract-model-set`, and methods for encoding, such as `encode-component` and `compile-project`.

These set of metalevel methods allows to work at three different levels: knowledge, code, and execution.

One of the capabilities of MetaKit is that allows to design meta-projects covering several verification strategies and to apply them to object-projects. An example of a meta-project performing detection and correction of other projects is a meta-project looking for missing or incompletely described relations between models, tasks and methods.

On another hand, MetaKit offers also capabilities for completing missing components of the current project from comparing the current project to some older fragments of designs and deducing what has to be added to the current project. Is in this approach how machine learning techniques could be used in MetaKit. An example of the incorporation of decision tree learning algorithms is described in [Slodzian, 1994a].

Finally, MetaKit can be used for controlling the execution. The execution control is based on a description of temporal and causal relations between components.

2.4 Introspective learning

Learning methods in a reflective framework are a type of inference methods that have a model of certain aspects of the system (the *self-model*) that is useful for improving the system behavior. Usually, this self-model is amenable to be analyzed in order to detect the failures and successes of the system. Moreover, the learning method has to have some knowledge about how to assign blame for failures and merit for successes to components of the system (the *meta-theory* of the learning method). As a result, the meta-theory has to decide which actions are amenable to be effected to the system to improve it. For instance, assuring that successful methods will be used in appropriate situations (similar situations or classes of situations) and that failures will not be repeated (for similar situations or classes of situations). This knowledge is also part of the meta-theory and constitutes the extension of the self-model that the reflection process will translate into effective modifications of the system.

The reification process and the type of model it constructs determines the kind of learning that can be performed. Usually, the models used in ML systems involve "sets of rules", in that learning results in the addition of new rules (and the modification of old rules) to the "rule set". Another important issue is the typology of situations for which learning is applied. In a "rule set" system, the typology is usually "false positives" (a rule fired and achieved an erroneous state) and "false negatives" (no rule fired that achieved the correct state); then, repair is achieved through learning of new rules or generalizing old rules for false negatives and deleting or specializing old rules for false positives. Using schema languages, on the other hand, the meta-theory talks about correct and incorrect retrieval of schemas and repair is realized modifying the indexing of schemas (see [Ram et al., 1992] for an instance of introspective learning in schema languages).

Learning methods that learn plans, such as learning methods in SOAR [Newell, 1990] and PRODIGY [Carbonell et al., 1991], have self-models about the applicability and ordering of operators. The meta-theory has to know about properties of the architecture and of the learning method in order to decide how to modify the rule-set. Since PRODIGY knows that new rules learned by EBL module are assured to be correct by EBL properties, PRODIGY knows new rules can be added and will not be modified or retracted. In SOAR, however, since chunking can overgeneralize, the reflection process has not only to construct new rules, but has to assure that old, overgeneralized rules will be less preferred to the new ones (adding a worse-preference between the two, since it "knows" that the semantics of the system will never execute the less preferred one when the preferred one is applicable).

Although this review is very short, the points we wish to make is that the learning methods, albeit implicitly, can be used only because they realize this knowledge about the architecture (the self-model). Another, collateral argument for this claim, is the fact that most ML "methods" have to be always modified or adapted to be usable in other domains where other systems are used. This is usually conceived of as "implementation details", but we argue this is not so: the proliferation of learning methods proves that some fundamental issue is at stake. Our claim is that the families and variations of ML methods, come from the fact that the (implicit) self-model is an essential part of learning, and many variants of a method come from variations of meta-theory and self-model. For instance, adapting a ML method to a different architecture require changes in the meta-theory to include what the learning component needs to know of the new system (and similarly adapting a ML method to some new domain involves adapting its meta-theoretic content to the features required by the new domain).

2.5 Conclusions

In this chapter we have reviewed the research relevant to our work organizing the presentation on four main topics: the research on knowledge modeling frameworks, the research on integrated architectures, the research on reflective representation languages, and the role of reflection and learning.

Knowledge modeling frameworks propose methodologies for analyzing and describing applications at a knowledge level in an implementation independent way. These methodologies are based on the task/method decomposition principle and on the analysis of knowledge requirements for problem solving methods. Usually, the task/method decomposition and the models resulting from the knowledge modeling analysis are acquired using learning techniques. Nevertheless, learning techniques are not incorporated for improving the behavior of the knowledge system from reasoning about the experience of acquired in solving problems. Moreover, the task/method decomposition and the models are fully determined in the knowledge modeling analysis. Another difficulty with some knowledge modeling methodologies is that knowledge models are not directly operationalizable.

Integrated architectures propose different frameworks for integrating problem solving and learning. A difficulty with integrated architectures is that their representation languages are far from knowledge modeling frameworks (for instance, SOAR representation language is based on productions). We have described three different architectures: THEO, SOAR, and PRODIGY. All these architectures integrate diverse learning mechanisms but present some limitations in their integration. THEO has fixed learning strategy based on three predefined learning mechanisms. Chunking is the only architecturally supported learning mechanism of SOAR and other learning techniques are difficult to integrate. PRODIGY integrates six learning modules but a specific learning module cannot influence nor take advantage of the learning form other modules.

Reflective representation languages propose different languages providing reflective mechanisms for describing and implementing knowledge systems that reason about themselves. These introspective mechanisms allows to develop complex knowledge systems in a more clear way.

Finally, we have discussed how the reification process and the type of model used in a system determines the kind of learning that can be performed.

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.

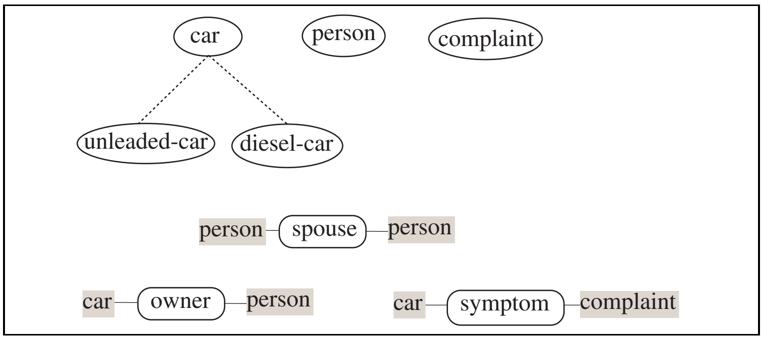


Figure 3.1. Part of the domain ontology of the diagnosis of car malfunctions application.

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

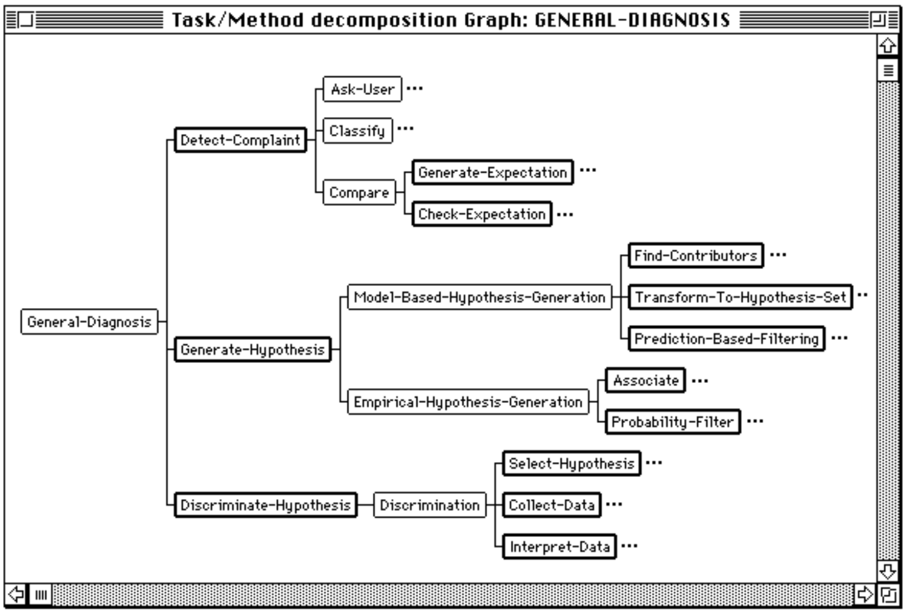


Figure 3.2. A browser of the task/method decomposition for a general diagnosis method. Methods are drown with thin boxes; tasks are drown with thick boxes; dots indicate not expanded terms.

[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

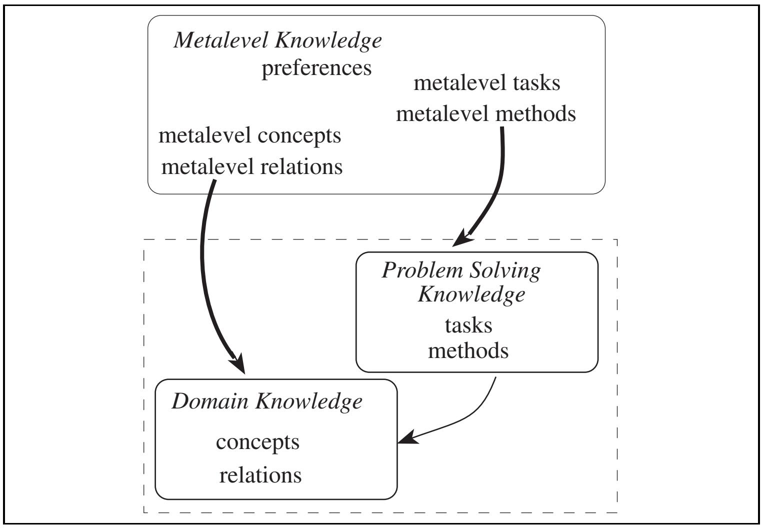


Figure 3.3. The Noos modeling framework.

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

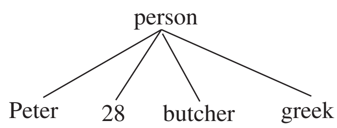
$$\texttt{person}(x_1, x_2, x_3, x_4)$$

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:

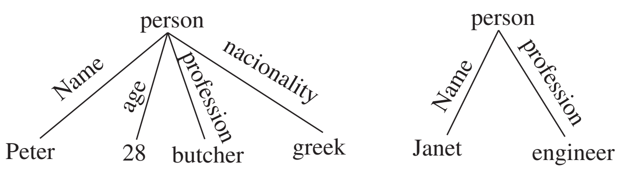


Figure 3.4. Labeled graph representation of two feature terms.

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):

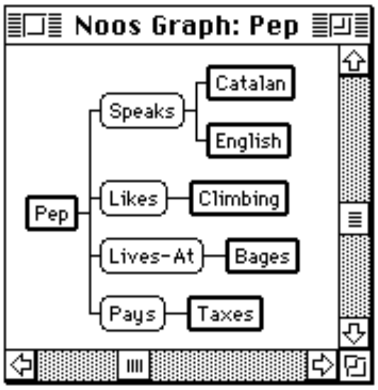


Figure 3.6. A Noos browser visualizing the labeled graph representation of Pep. Feature names are represented as thin boxes. Note that feature terms are rooted labeled graphs. The root node in this example is node Pep.

- 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.

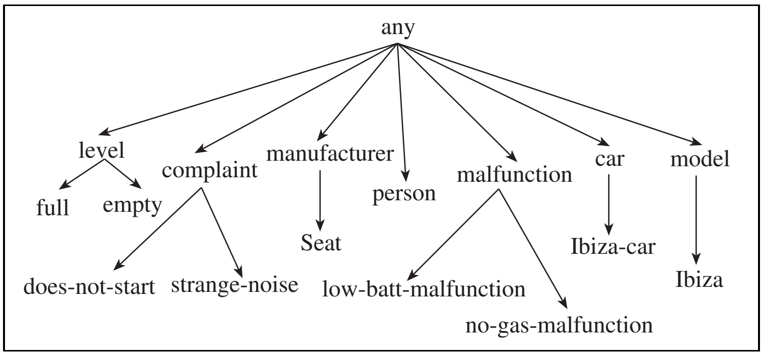


Figure 3.8. The Noos sort hierarchy used in the example of Fig. 3.7 (the diagnosis of car malfunctions domain).

`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)
  (brothers Adam Arthur)
  (sisters Abigail Alison))

(define (person Abigail)
  (husband Bob))

(define (person Alison)
  (husband Bart))

(define (person Adam)
  (wife Linda))

(define (person Arthur)
  (wife Lucy))

(define (person Linda)
  (brothers Clement Charles))

(define (person Lucy)
  (brothers David Douglas))
```

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

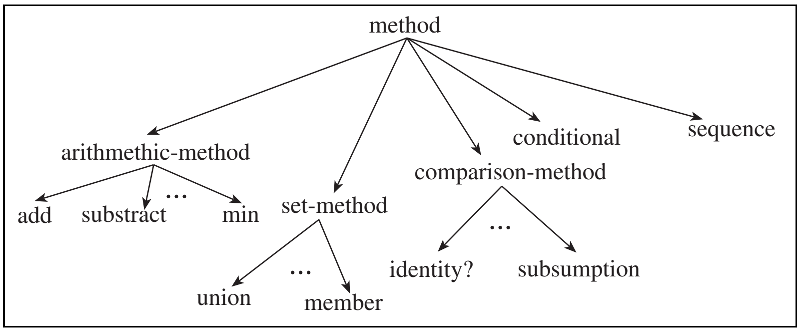


Figure 3.10. A partial expanded tree of the Noos built-in method hierarchy.

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*`)))`

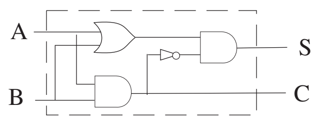
For instance, in Figure 3.7 the built-in `identity?` method is used to describe the `empty-level?` feature value of `car`. The `empty-level?` feature value will be `true` when `gas-level-in-tank` is `empty` and `false` otherwise.

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)))))
```



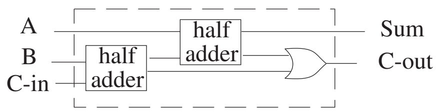


Figure 3.11. Definition of an adder circuit.

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-

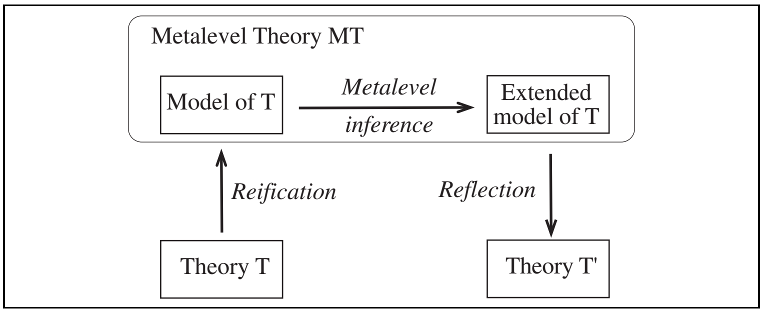


Figure 3.13. The reflection cycle. *Reification* constructs a model of a theory T. *Metalevel inference* infers new facts or takes new decisions that extend (or modify) the model of T using a meta-theory MT. Finally, *reflection* constructs a new theory T' that complies the extended model of T.

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.

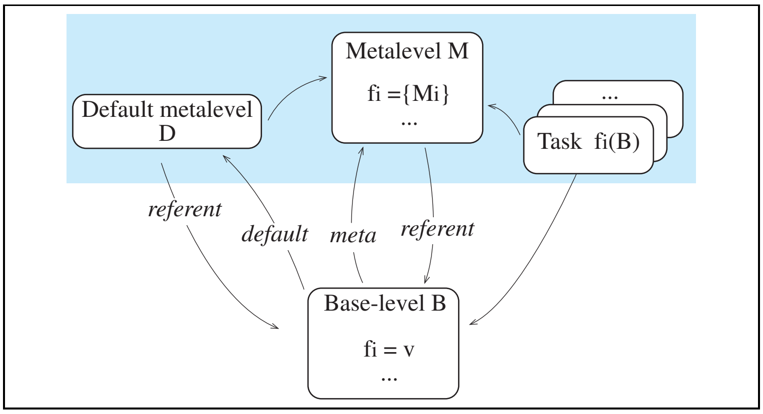


Figure 3.14. Metalevel components of Noos and their causal connections.

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.

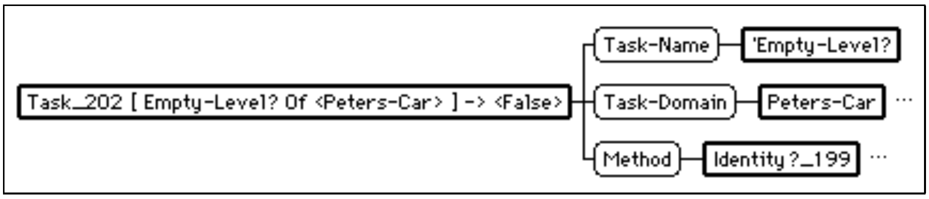


Figure 3.15. Task feature term reifying the inference of feature `empty-level?` of `Peters-Car`. Note that the printname of the task contains the referent.

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,

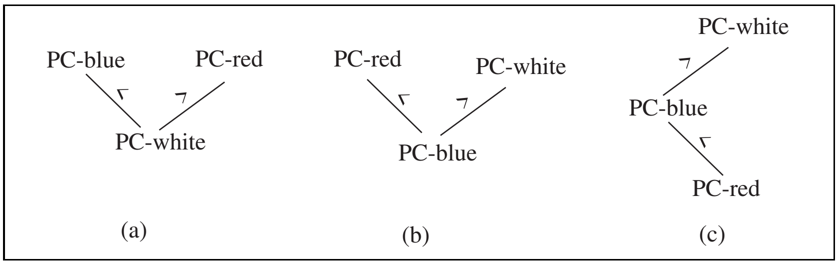


Figure 3.17. Graphical representation of three different preferences over computers. (a) corresponds to `cheaper-pref`, (b) is built using `disk-pref`, and (c) is obtained with `faster-pref`.

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).

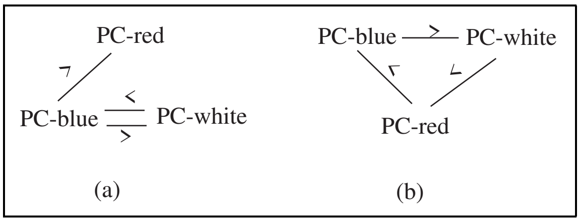


Figure 3.18. Combining preferences.

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.

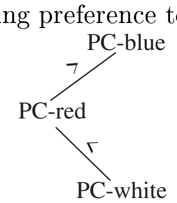
Higher order preferences

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

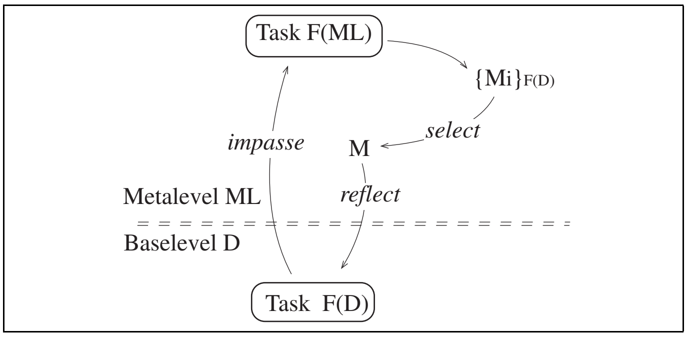


Figure 3.19. Solving and *Impasse* for a task `F(D)` in the metalevel `ML` of `D`.

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 F_i(M) is engaged (see first kind of backtracking). Finally, when backtracking on the first subtask F_1(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 F_1(M), ..., F_m(M) are achieved. Then, backtracking is engaged for inferring another value for the last subtask F_m(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 $\mathtt{F}_1, \mathtt{F}_2, \ldots, \mathtt{F}_n$ and tasks $\mathtt{F}_1\mathtt{(M)}, \mathtt{F}_2\mathtt{(M)}, \ldots, \mathtt{F}_n\mathtt{(M)}$ engaged for inferring the value of the required features,

1. Backtracking is engaged in task $\mathtt{F}_n\mathtt{(M)}$ using the *Next-Value* process.
2. Backtracking on task $\mathtt{F}_n\mathtt{(M)}$ can succeed or can fail: when it succeeds, the evaluation of method `M` is resumed to next step; when it fails, backtracking on task $\mathtt{F}_{n-1}\mathtt{(M)}$ is recursively engaged (see *Noos-Eval* process). When backtracking on the first subtask $\mathtt{F}_1\mathtt{(M)}$ fails, the *Eval-Next* process fails.
3. When all tasks $\mathtt{F}_1\mathtt{(M)}, \mathtt{F}_2\mathtt{(M)}, \ldots, \mathtt{F}_n\mathtt{(M)}$ 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 $\mathtt{F}_1\mathtt{(M)}, \mathtt{F}_2\mathtt{(M)}, \ldots, \mathtt{F}_n\mathtt{(M)}$ 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.

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.

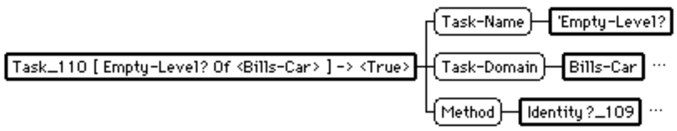
Once a problem task is solved Noos automatically memorizes (stores and indexes) the episodic model that has been built.

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`

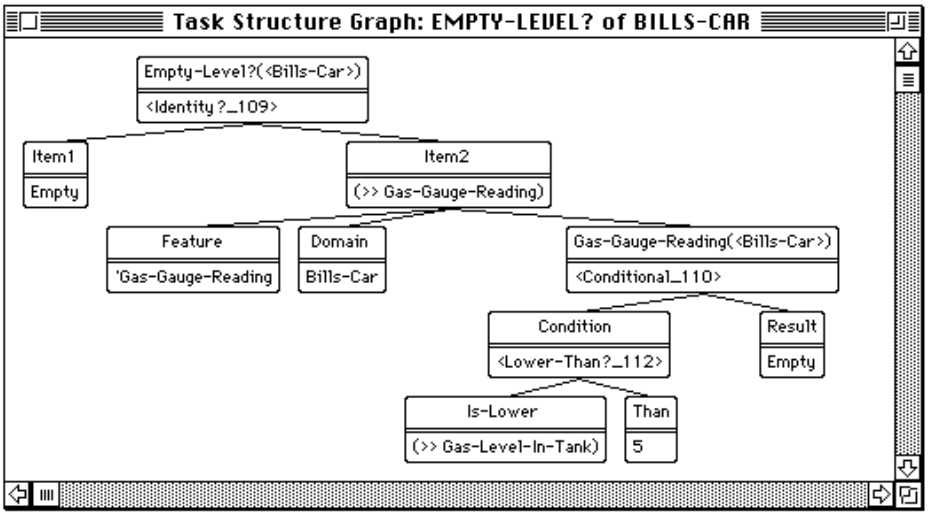


Figure 4.1. A browser of the task/method decomposition from the episodic model of the `empty-level?(Bills-car)` problem task.

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

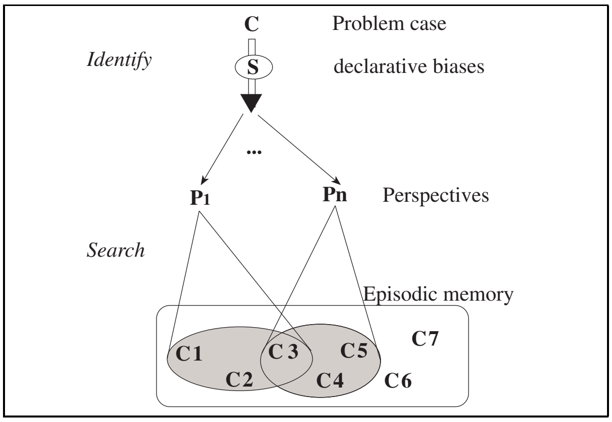


Figure 4.2. Using perspectives in a retrieval task. First, given a problem case C and using syntactic patterns S as declarative biases, the relevant aspects of C are determined and perspectives $P_1 \cdots P_n$ are built. Later, these perspectives are used in retrieval methods to search precedent cases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

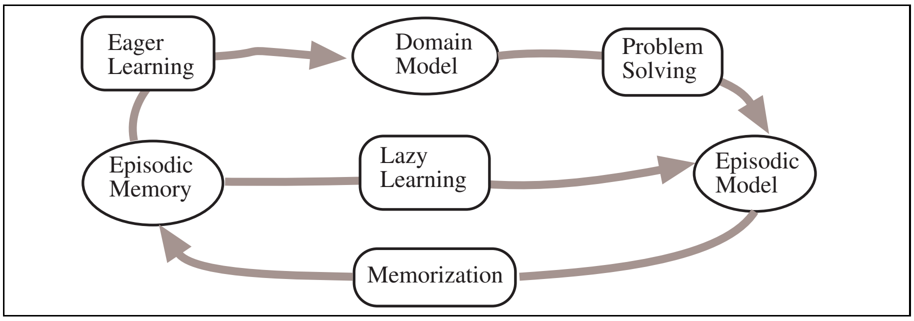


Figure 4.3. Lazy and eager learning and the construction of domain models and episodic models.

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.

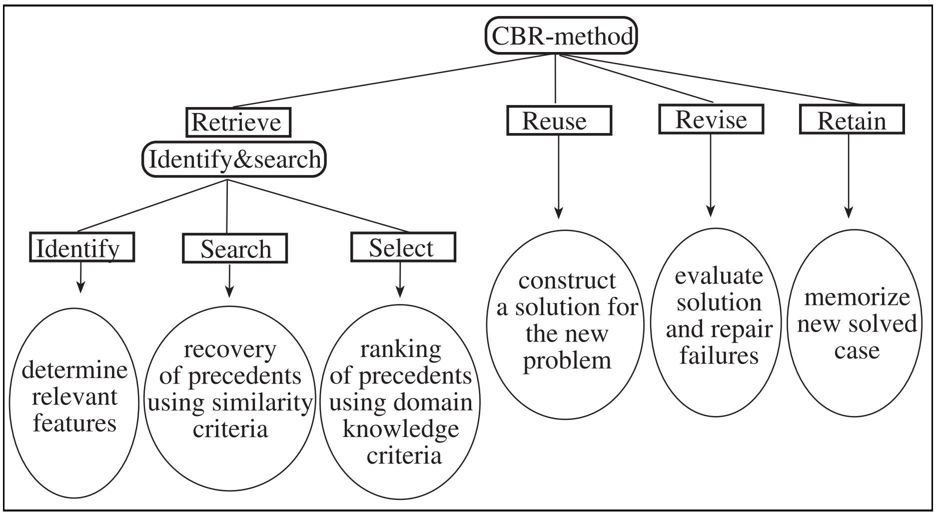


Figure 4.4. Task/method decomposition of Case-based reasoning methods.

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).

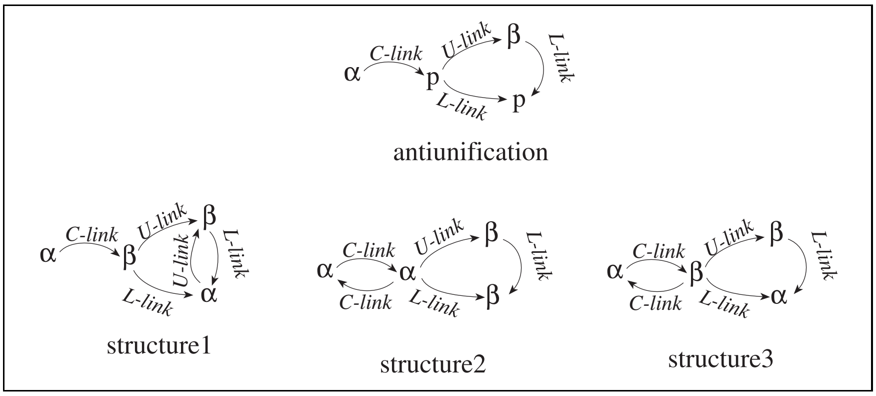


Figure 4.6. Induction example.

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 $\texttt{M}_{op}$ 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.

Chapter 5

Noos Formalization

The goal of this Chapter is to present a formal description of the Noos representation language. We will present the Noos formal syntax based on feature terms, its semantics, and the formal model of the Noos inference process.

Our approach to formalize Noos syntax and semantics is related to the research based on λN calculus [Dami, 1994], *ψ-terms* [Aït-Kaci and Podelski, 1993] [Carpenter, 1992] [Backofen and Smolka, 1995], and extensible records [Cardelli and Mitchell, 1994] that propose formalisms to model object-oriented programming constructs.

As we have stated in Chapter 3 the Noos representation language is based on feature terms formalism. 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 two properties can be achieved by building terms with parameters identified by name (regardless of order or position) and with no fixed number of parameters.

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*).

The λN calculus formalism is an extension of λ-calculus that introduces named parameters as arguments of lambda abstractions. In λN, a lambda abstraction can have multiple arguments which can be bound separately and in any order. λN provides a clear way to model extensible and recursive records, and also offers a clear mapping among functions and records. We use λN calculus to provide a syntax for Noos feature terms. Moreover, λN calculus capabilities for modeling extensible knowledge are used for modeling the refinemement mechanism of Noos. λN calculus lexical scoping is used for modeling path references and path equality.

The *ψ-term* formalism offers an alternative approach to model relational and object-oriented programming. [Aït-Kaci and Podelski, 1993] presents a semantical interpretation for *ψ-terms* that allows three equivalent representations: terms, clauses, and graphs. *ψ-term* calculus is based on the notions of unification and subsumption. We adopt a related approach to the semantical interpretation of ψ-terms in order to provide a semantical interpretation of Noos feature terms. Following the *ψ-term* formalism, feature terms are interpreted as partial descriptions. This semantical interpretation of feature terms brings an ordering relation among them. We call this ordering relation *subsumption*. The intuitive meaning of subsumption is that of *informational ordering*. We say that a feature term ψ subsumes another feature term ψ' (noted $\psi \sqsubseteq \psi'$) when all information in ψ is also contained in ψ'.

Last elements of the Noos language are the metalevel relation among feature terms, reflection, and the inference process involved in solving a specific problem task. λN calculus and ψ-terms are not the best suited formalisms for modeling the Noos inference process. We formally describe the global inference process in our system using Descriptive Dynamic Logic [Sierra et al., 1996]. Descriptive Dynamic Logic (DDL) is a propositional dynamic logic (PDL) [Harel, 1984] that provides a general framework for describing and comparing reflective knowledge systems. DDL models knowledge systems as a set of units with initial local theories written in possibly different languages. Each unit is also usually allowed to have its own intra-unit deductive system. Moreover, the whole knowledge system is equipped with an additional set of deductive rules, called *bridge rules*, to control the information flow among the different units of the knowledge system. Thus, the DDL approach is very useful to model reflective systems based on the use of several units containing local theories (or meta-theories acting upon theories) that influence and/or modify each other.

Moreover, two specific elements of Noos have to be defined formally: `preferences` and `perspectives`. We presented in Section 3.4 a declarative mechanism for decision making about sets of alternatives we call preferences. Reasoning with preferences is modeled by partially ordered sets with a set of operations for constructing new preferences and combining them.

Perspectives, explained in Section 4.3, are a mechanism to describe declarative biases for retrieval in the Noos episodic memory. Using feature terms, perspectives are formalized as second order feature terms that denote sets of terms.

The structure of this Chapter is as follows: Section 5.1 contains a brief introduction to λN calculus. Sections 5.2 to 5.9 present the syntax and semantics of feature terms. Section 5.10 describe the formal basis for reasoning with preferences in Noos. Section 5.11 presents the formal principles on which perspectives are based. Section 5.12 contains a brief introduction to DDL. Section 5.13 describes the inference in Noos using the DDL formalism. Finally, Section 5.14 summarizes our approach to formalize the Noos representation language.

5.1 Basic notions of λN calculus

The lambda calculus with Names (λN calculus) is an extended λ-calculus developed by Laurent Dami [Dami, 1994] that introduces named parameters as arguments of lambda abstractions. λN uses Bruijn indices [de Bruijn, 1972] as the basis for the introduction of named parameters. In λN, a lambda abstraction can have multiple arguments which can be bound separately and in any order. This feature brings what Dami calls the *extensibility property*: existing software fragments can be augmented with new features while remaining compatible with the original contexts in which they were used. The extensibility property of λN calculus allows to model records, extensible datatypes, and object-oriented programming constructs. Moreover, named parameters are introduced in λN without affecting the semantics of functions because names can be totally dismissed in the Bruijn calculus. Thereby, λN formalism preserves results of the standard lambda calculus such as confluence.

The λN calculus is constructed from a set $\mathcal{V}$ of variables (or labels). Using this set of variables and extending the λ-calculus syntax, the set of terms Λ_N of the λN calculus is built from the following basic syntax[1]:

$$\begin{array}{lcl} \mathrm{a} & ::= & \lambda(x_1 \ldots x_n)\mathrm{a} \mid \mathrm{a}(x \to \mathrm{b}) \mid \mathrm{a}! \mid \mathrm{v} \\ \mathrm{v} & ::= & x \mid \backslash\mathrm{v} \end{array}$$

Figure 5.1. Basic λN syntax.

In λN notation, a lambda abstraction $\lambda(x_1 \ldots x_n)\mathrm{a}$ can have several arguments—identified by a named parameters and declared as a list of variables $x_1 \ldots x_n$ inside two parentheses—which can be bound separately and in any order. As a consequence of this approach, the functional application operation of λ-calculus has to be split in two different operations, called *bind* and *close* operations. An expression of the form $a(x \to b)$ (called *bind expression*) binds b to the parameter x in the abstraction a. An expression of the form $a!$ (called *close expression*) ends a sequence of bind expressions.

In the context of λN, the usual β-reduction rule of lambda calculus is split in two *lambda-bind* and *lambda-close* rules:

$$(\lambda(x_1 \ldots x_n)\mathrm{a})(x_i \to \mathrm{b}) \quad \longrightarrow_\beta \quad \lambda(x_1 \ldots x_n)\mathrm{a}[x_i := \mathrm{b}] \tag{5.1}$$

$$(\lambda(x_1 \ldots x_n)\mathrm{a})! \quad \longrightarrow_\beta \quad a[x_* := \mathbf{err}] \tag{5.2}$$

where x_* denotes any unbound variable. The **err** constant is represented in λN as a lambda abstraction that always reduces to itself—and that in object-oriented systems corresponds to the "message not understood" error.

The main difference with λ-calculus is that the *lambda-bind* rule (5.1) performs a substitution without removing the outermost abstraction level (the 'λ'), while the *lambda-close* rule (5.2) removes the 'λ' and substitutes any remaining unbound variables by **err** (the constant representing run-time errors).

Lexical scoping of variables is treated in the following way: a local declaration (a variable declared as a parameter of a lambda abstraction) takes precedence

[1]Notation remark: we will use x to denote variables; and $\mathrm{a}, \mathrm{b}, \mathrm{c}, \ldots$ to denote terms.

over the declaration of a variable with the same name in outer lambda abstractions. In case the same name is used at different lambda abstraction levels, λN provides a scope escape operator '\' (backslash). The use of n occurrences of this escape operator preceding a variable x specifies that the corresponding declaration of x will be looked for ignoring the inner n abstraction levels. For instance, consider the expression

$$\lambda(xy)\lambda(xz)x + y + z + \backslash x$$

and assume that infix addition is part of the language; then x and $\backslash x$ are references to different variables. The first x reference is bound to the x parameter defined in the inner lambda abstraction $\lambda(xz)$, while $\backslash x$ reference is bound to the x parameter defined in the outer lambda abstraction $\lambda(xy)$.

Recursion is defined in λN by means of a fixed point operation. Specifically, a fixed point operation over a functional $\lambda(x)a$ corresponds to the usual combinator Y in the pure lambda calculus. An expression with recursion over parameter x is written $\mu(x)a$ and the translation $\mathbf{T}$ into de basic syntax is the following:

$$\begin{array}{lcl} \mathbf{T}(\mu(x)a) & = & Y_x(x \rightarrow \lambda(x)a) \\ Y_x & = & \lambda(x)(\lambda(x)\backslash x(x \rightarrow x(x \rightarrow x)!)!)(x \rightarrow (\lambda(x)\backslash x(x \rightarrow x(x \rightarrow x)!)!)) \end{array}$$

Extensible records

From the basic λN syntax, *extensible records* can be introduced in the λN formalism [Dami, 1994]. Extensible records are written with braces; fields are written with the $\doteq$ symbol; field selection uses the common dot notation. The syntax of these constructs and the translation $\mathbf{T}$ into the basic syntax is the following:

$$\begin{array}{lcl} \mathbf{T}(\{x_1 \doteq a_1 \dots x_n \doteq a_n\}) & = & \lambda(sel)sel(x_1 \rightarrow a_1)\dots(x_n \rightarrow a_n) \\ \mathbf{T}(a.x) & = & a(sel \rightarrow \lambda(x).x)!! \end{array}$$

A record is translated to a function which takes a selector *sel* and binds all fields to corresponding named parameters in that selector. A selector for field x is just an identity function on that name, so a field selection operation simply binds the appropriate selector to the *sel* argument of the record. Notice that the field selection operation is defined in basic syntax as an identity function plus two close operations. The first close operation is used to close the bindings of record a, while the second close operation binds the appropriate selector to the *sel* argument of the record.

Next, a record concatenation operation '$\ll$' can be introduced providing an incremental mechanism of record construction. The $a \ll b$ expression yields a new record from a concatenation of two records a and b in a right-preferential order. That is to say, fields declared in b override the fields also declared in a. The correspondent translation into the basic λN syntax is the following:

$$\mathbf{T}(a \ll b) = a(sel \rightarrow \mathbf{T}(b))!$$

For instance, the following expression:

$\{x \doteq 1 \quad y \doteq 2\} \ll \{x \doteq 4 \quad z \doteq 5\}$

yields a new record with three fields x, y, z containing the values 4,2,5 respectively:

$\{x \doteq 4 \quad y \doteq 2 \quad z \doteq 5\}$

Recursion can be used to define recursive records. The syntax of records is enlarged for recursive records as follows:

$$\mu(rec)\{x_1 \doteq a_1 \dots x_n \doteq a_n\}$$

For instance we can define the following recursive record

$$\begin{array}{lll} Seasons = \mu(rec) & \{ & spring \doteq \{name \doteq \text{“spring”} \quad next \doteq rec.summer\} \\ & & summer \doteq \{name \doteq \text{“summer”} \quad next \doteq rec.autumn\} \\ & & autumn \doteq \{name \doteq \text{“autumn”} \quad next \doteq rec.winter\} \\ & & winter \doteq \{name \doteq \text{“winter”} \quad next \doteq rec.spring\} \\ & \} & \end{array}$$

and, for example, $Seasons.summer.next.next.name$ yields “winter”.

When we define recursive records at different levels, the ‘\’ scope escape operator allows references to these different levels. Using the ‘\’ notation, we can compact the syntax for describing recursive record structures by removing the explicit reference to rec. For instance, the previous $Seasons$ recursive record can be described using compact notation as follows:

$$\begin{array}{lll} Seasons = & \{ & spring \doteq \{name \doteq \text{“spring”} \quad next \doteq \backslash\backslash summer\} \\ & & summer \doteq \{name \doteq \text{“summer”} \quad next \doteq \backslash\backslash autumn\} \\ & & autumn \doteq \{name \doteq \text{“autumn”} \quad next \doteq \backslash\backslash winter\} \\ & & winter \doteq \{name \doteq \text{“winter”} \quad next \doteq \backslash\backslash spring\} \\ & \} & \end{array}$$

Given the basic notions about λN calculus, next we describe the translation rules from Noos descriptions to λN calculus.

5.2 Noos formal syntax

In this section we will use the λN formalization of extensible records as a basis to formalize Noos descriptions as feature terms. The Noos refinement mechanism will be formalized by means of the record concatenation operation.

There are two types of feature terms in Noos: *constant feature terms* and *evaluable feature terms*. Feature terms are defined on a signature Σ composed by a set of sort names, a set of method names, and a set of feature names. Formally,

Definition 5.1 *(Noos Signature)*
We define the Noos *signature as the tuple* $\Sigma = \{\langle \mathcal{S}, \leq_s \rangle, \langle \mathcal{M}, \leq_m \rangle, \mathcal{F}\}$ *such that:*

- $\mathcal{S}$ *is a finite set of sort symbols including* $\perp_s$ *and* $\leq_s$ *is an order relation such that* $\perp_s$ *is the smallest element.*
- $\mathcal{M}$ *is a finite set of method symbols including* $\perp_m$ *and* $\leq_m$ *is an order relation such that* $\perp_m$ *is the smallest element.*
- $\mathcal{F}$ *is a finite set of* feature symbols*;*

Given the Noos signature Σ, we define the set Γ of feature terms as a union of constant feature terms and evaluable feature terms:

$$\Gamma = \Gamma_c \cup \Gamma_m$$

In the next section we will introduce Noos formal syntax for constant feature term step to step. The complete description of the formal syntax of Noos is given in Figure 5.2. Evaluable feature terms will be introduced in Section 5.9.

5.2.1 Constant feature terms

Given the signature Σ, we define *constant feature terms* as follows[2]:

Definition 5.2 *A constant feature term* ψ *is an expression of the form:*

$$\psi \quad ::= [s\ f_1 \doteq \mathrm{fv}_1 \cdots f_n \doteq \mathrm{fv}_n]$$

where s *is a sort in* $\mathcal{S}$*,* $f_1, \cdots, f_n$ *are pairwise distinct features in* $\mathcal{F}$*,* $n \geq 0$*, and each* fv_i *is a feature value.*

A feature term is translated to λN syntax, using the translation rule $\mathbf{T}$, as a recursive record as follows:

$$\mathbf{T}(s\,[f_1 \doteq \mathrm{fv}_1 \cdots f_n \doteq \mathrm{fv}_n]) \ = \ \{f_1 \doteq \mathrm{fv}_1 \cdots f_n \doteq \mathrm{fv}_n\}$$

The sort of a feature term (noted $\tau(\psi) = s$) is a component not represented in the λN notation and will be explained in Section 5.2.2.

Feature values

There are three kinds of feature values:

$$\begin{array}{rcl} \mathrm{fv} & ::= & \psi \\ & | & \psi_1 \cdots \psi_n \\ & | & \mathrm{ref} \end{array}$$

where the first one is a term; the second is a set of terms; and the third is a path reference (described in Section 5.2.3). We have shown in Section 3.2.4 that feature values can also be described using closed methods, and in Section 5.9 we will extend the syntax for incorporating closed methods.

[2]We use two equivalent notations for feature terms: the first one is as an horizontal enumetarion of features (the notation used here). The second notation is the vertical enumeration of features (useful to represent feature terms inside feature terms).

5.2.2 The sort of a feature term

From the set Γ of feature terms we define a sorting function $\tau : \Gamma \rightarrow \mathcal{S} \cup \mathcal{M}$, assigning either a sort symbol or a method symbol to each feature term. A constant feature term has assigned a sort symbol. An evaluable feature term has assigned a method symbol. Specifically, given a feature term ψ defined as follows,

$$\psi \ = \ [s \ f_1 \doteq \mathrm{fv}_1 \cdots f_n \doteq \mathrm{fv}_n]$$

the sorting function τ for ψ is defined as $\tau(\psi) = s$. We will say that the sort of ψ is s.

5.2.3 Path references

A path reference has two components: a *head* and a *path*. A head is constructed using the backslash notation (for instance, '\\') A path is constructed as a concatenation of field selections using the dot notation (for instance '$f_1.f_2$'). The formal syntax for path references is the following:

$$\begin{array}{lcl} \mathrm{ref} & ::= & \mathrm{head} \mid \mathrm{head\ path} \\ \mathrm{head} & ::= & \backslash\mathrm{head} \quad | \quad \backslash \\ \mathrm{path} & ::= & f.\mathrm{path} \quad | \quad f \end{array}$$

An example of a path reference is $\backslash\backslash f_1.f_2$. A path reference may have only a sequence of backslashes (for instance \\). Path references are used to provide a formalization of Noos relative path references and a precise interpretation of the rules of scope and refinement (see Section 5.3).

Reference over sets

Since feature values can be sets, path references have to deal with feature values that are sets. The field selection operator over sets is defined as the pointwise extension of the λN field selection operator. The result of a field selection over a set is a new set. Therefore, we define the field selection operator over sets as follows:

$$S.f \equiv \bigcup_{s \in S} s.f$$

5.2.4 Refinement

Feature terms are always constructed in the Noos language by refinement. The definition of a feature term ψ by refinement of another feature term ψ' will be formalized as a record concatenation as follows:

$$\psi \quad ::= \quad \psi' \ \ll \ [s \ f_1 \doteq \mathrm{fv}_1 \cdots f_n \doteq \mathrm{fv}_n]$$

$$
\begin{array}{llll}
\psi & ::= & [s\ f_1 \doteq \text{fv} \cdots f_n \doteq \text{fv}] & ;\ \text{feature term} \\
 & | & \psi \ll [s\ f_1 \doteq \text{fv} \cdots f_n \doteq \text{fv}] & ;\ \text{refinement} \\
 & & & \\
\text{fv} & ::= & \psi & ;\ \text{record} \\
 & | & \psi_1 \cdots \psi_n & ;\ \text{set of records} \\
 & | & \text{ref} & ;\ \text{reference} \\
 & & & \\
\text{ref} & ::= & \text{head} \mid \text{head path} & ;\ \text{path reference} \\
 & & & \\
\text{head} & ::= & \backslash\text{head} \quad | \quad \backslash & ;\ \text{outer reference} \\
 & & & \\
\text{path} & ::= & f.\text{path} \quad | \quad f & ;\ \text{path}
\end{array}
$$

Figure 5.2. Formal syntax of Noos using the λN approach.

$$
\begin{array}{lll}
\Upsilon(\texttt{desc}) & = & \Upsilon_d(\texttt{desc}, 0) \\
 & & \\
\Upsilon_d(\texttt{(define (const)} & = & \Upsilon(\texttt{const}) \ll [\mathit{const}\ \Upsilon_f(\texttt{fdesc}_1, \ell) \\
\quad \texttt{fdesc}_1 \cdots \texttt{fdesc}_n\texttt{)}, \ell) & & \quad \cdots \Upsilon_f(\texttt{fdesc}_n, \ell)] \\
 & & \\
\Upsilon_f(\texttt{(f desc)}, \ell) & = & \texttt{f} \doteq \Upsilon_d(\texttt{desc}, \ell + 1) \\
\Upsilon_f(\texttt{(f desc}_1 \cdots \texttt{desc}_n\texttt{)}, \ell) & = & \texttt{f} \doteq \Upsilon_d(\texttt{desc}_1, \ell + 1) \cdots \Upsilon_d(\texttt{desc}_n, \ell + 1) \\
\Upsilon_f(\texttt{(f ref)}, \ell) & = & \texttt{f} \doteq \Upsilon_r(\texttt{ref}, \ell + 1) \\
 & & \\
\Upsilon_r(\texttt{(>> f}_1 \cdots \texttt{f}_n\texttt{)}, \ell) & = & \underbrace{\backslash \cdots \backslash}_{\ell} f_n. \cdots .f_1
\end{array}
$$

Figure 5.3. Translation rules from Noos syntax to λN syntax.

where ψ is defined with sort s ($\tau(\psi) = s$) and by extending ψ' with $f_1, \ldots, f_n$ features.

Note that concatenation operation $\ll$ is a right-preferential order operation. This means that all the definitions of features $f_1, \ldots, f_n$ also defined in ψ' override the definitions given in ψ'.

5.3 Translation rules from Noos to λN

We have introduced the formal syntax of Noos that is fully described in Figure 5.2. Now, we will introduce the translation rules Υ from Noos syntax to Noos formal syntax. There are three kinds of translation rules: translation rules for descriptions Υ_d, translation rules for features Υ_f, and translation rules for references Υ_r. We will explain these translation rules incrementally. A complete description of translation rules is given in Figure 5.3.

The Noos language allows the use of names for describing feature values. In Noos formal syntax feature terms have no name. Therefore, before translating a given description D to the formal syntax, we have to perform a preprocess where each name reference is substituted by either the description that it denotes or by a path reference to another feature with the same name reference as feature value. The specification of the same name reference as the feature value of different features requires the satisfaction of the *path equality* property (see Section 3.2.3). In order to preserve this path equality only one occurrence of a name reference will be translated to the description that it denotes. In the rest of occurrences the name reference will be translated to a path reference to the first occurrence.

Translation rules have two parameters. The first parameter contains a description (the text to be translated). The second parameter holds and integer indicating the depth level of the description according to the root description. The depth level will be used to translate path references.

A Noos description `desc` is translated with a description rule starting with depth level zero.

$$\Upsilon(\texttt{desc}) \;=\; \Upsilon_d(\texttt{desc}, 0)$$

Since a feature term is built by refinement of another feature term, a description is translated as the concatenation of two records applying the following rule:

$$\begin{array}{lcl} \Upsilon_d(\texttt{(define (const)} & = & \Upsilon(\texttt{const}) \ll [\mathit{const}\ \Upsilon_f(\texttt{fdesc}_1, \ell) \\ \quad\quad \texttt{fdesc}_1 \cdots \texttt{fdesc}_n\texttt{)}, \ell) & & \quad\quad\quad\quad \cdots \Upsilon_f(\texttt{fdesc}_n, \ell)] \end{array}$$

where the feature term is built concatenating (refining) the feature term denoted by `const` ($\Upsilon(\texttt{const})$) with a record composed by the features defined in the body of the description translated using translation rules for features Υ_f.

Given the description `fdesc` of a feature as the pair `(f v)`, translation rules Υ_f define a field of a record with name `f` and with value that obtained from the translation of `v` to formal syntax increasing the depth level by one. There is one translation rule Υ_f for each kind of feature value:

$$\begin{array}{lcl} \Upsilon_f\,(\texttt{(f desc)}, \ell) & = & \texttt{f} \doteq \Upsilon_d(\texttt{desc}, \ell+1) \\ \Upsilon_f\,(\texttt{(f desc}_1 \cdots \texttt{desc}_n\texttt{)}, \ell) & = & \texttt{f} \doteq \Upsilon_d(\texttt{desc}_1, \ell+1) \cdots \Upsilon_d(\texttt{desc}_n, \ell+1) \\ \Upsilon_f\,(\texttt{(f ref)}, \ell) & = & \texttt{f} \doteq \Upsilon_r(\texttt{ref}, \ell+1) \end{array}$$

New (sub)descriptions are translated using Υ_d rules. Path references are translated using the following Υ_r rule:

$$\Upsilon_r\,(\texttt{(>> f}_1 \cdots \texttt{f}_n\texttt{)}, \ell) \;=\; \underbrace{\backslash \cdots \backslash}_{\ell} f_n. \cdots .f_1$$

Note that a Noos path reference such that `(>> father mother)` is built as a concatenation of field selections in reverse order, i.e. `mother.father`, and containing in the head as many backslashes as the depth level.

Absolute path references are not allowed in the formal language. This means that they have to be translated to relative path references. There are two ways of translating absolute path references:

```
(define (Person :id Paul)
  (spouse Mary)
  (child (>> child of Mary)))

(define (Person :id Mary)
  (spouse Paul)
  (child (define (person)
           (age 3))))

(define (Person)
  (spouse (define (person)
            (spouse (>>))
            (child (define (person)
                     (age 3)))))
  (child (>> child of (>> spouse))))
```

Figure 5.4. Substitution of name references. Given two descriptions of `Paul` and `Mary`, `Paul` description is translated to the last description where the name references to `Mary` are substituted by its description in feature `spouse` and by a relative path reference in the absolute path reference `(>> child of Mary)` in feature `child`.

- A first case is when an absolute path reference `(>> F of D)` has a name reference `D` that it occurs as a feature value in another feature `F'`. In this case, the name reference `D` can be substituted to a relative path to the `F'` feature. Figure 5.4 shows an example of a substitution of an absolute path reference.

- A second is when an absolute path reference `(>> F of D)` has a name reference `D` that it does not occur as a feature value in another feature. In this situation, this name reference cannot be substituted by a path reference. We can extend the description with a new feature that has the name reference as value. Consequently, the name reference in the absolute path can now be substituted to a relative path to the new feature.

We are now ready to show, using an example, how a description in Noos syntax is translated to Noos formal syntax. Taking as example the previous description of `Peter`:

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

where `Peter` is defined by refinement of `Person` and the car that `Peter` is driving is defined by refinement of `Car`. Moreover, `Car` is defined as follows

```
(define Car
  (owner (define (person)))
  (gas-level-in-tank level)
  (gas-gauge-reading (>> gas-level-in-tank)))
```

and `person`, `does-not-start`, `full`, and `level` are defined with no features.

Then, the feature term that represents `Peter` description is built from the two following concatenations[3]:

$$\Upsilon(\texttt{Peter}) = \Upsilon(\texttt{Person}) \ll \begin{bmatrix} \mathsf{Person} & & \\ \mathit{age} & \doteq & \mathit{28} \\ \mathit{drives} & \doteq & \Upsilon(\texttt{Car}) \ll \begin{bmatrix} \mathsf{Car} & & \\ \mathit{owner} & \doteq & \backslash\backslash \\ \mathit{complaint} & \doteq & \mathit{does\text{-}not\text{-}start} \\ \mathit{gas\text{-}level\text{-}in\text{-}tank} & \doteq & \mathit{full} \end{bmatrix} \end{bmatrix}$$

Next, applying the operation of record concatenation to the description of that car we will obtain `gas-gauge-reading?` and `empty-level?` feature descriptions from the feature term `car`. Thus, after the concatenation operation, and since `person` is a feature term without features, we will obtain the following term:

$$\Upsilon(\texttt{Peter}) = \begin{bmatrix} \mathsf{Person} & & \\ \mathit{age} & \doteq & \mathit{28} \\ \mathit{drives} & \doteq & \begin{bmatrix} \mathsf{Car} & & \\ \mathit{owner} & \doteq & \backslash\backslash \\ \mathit{complaint} & \doteq & \mathit{does\text{-}not\text{-}start} \\ \mathit{gas\text{-}level\text{-}in\text{-}tank} & \doteq & \mathit{full} \\ \mathit{gas\text{-}gauge\text{-}reading} & \doteq & \backslash \mathit{gas\text{-}level\text{-}in\text{-}tank} \end{bmatrix} \end{bmatrix}$$

Notice that the reference in the `owner` feature is to `Peter` (two backslashes), and the reference in feature `gas-gauge-reading` is to the feature `gas-level-in-tank` in `car` (one backslash).

An additional notion that will be used later is the notion of *subterm* of a feature term:

Definition 5.3 *(Subterm)*
A feature term ψ_i is a subterm of a feature term ψ if ψ_i can be accessed from ψ following a path $f_1.\cdots.f_n$.

For instance, in the previous example of `Peter` feature term the `car` feature term is a subterm accessible from `Peter` by the path constituted by the `drives` feature.

[3]We write the sort assigned to a feature term in the head of the term in sans serif font, like Person or Car in the example.

5.4 Using variables in feature terms

Feture terms can also be represented using variables instead of path references. Variables, as described in [Aït-Kaci and Podelski, 1993], provide a notation based on tags. The motivation of using this new representation for feature terms is that variables provide a more adequate notation for defining the semantical interpretation of feature terms.

A feature term can be represented using variables by means of assigning a different variable to each feature term and replacing path references by the variable assigned to the referenced feature term. Specifically, a feature term, using variables, is defined as follows:

Definition 5.4 *Given the signature Σ and a set $\mathcal{V}$ of variables, we define a feature term ψ as an expression of the form:*

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

where X is a variable in $\mathcal{V}$; s is a sort in $\mathcal{S}$; $f_1, \cdots, f_n$ are pairwise distinct features in $\mathcal{F}$; $n \geq 0$; each Ψ_i is a set of feature terms and variables; and at most one occurrence of each variable is sorted.

We call the variable X in the above feature term the *root* of ψ (noted $Root(\psi) = X$), and say that X is *sorted* by the sort s (noted $\tau(X) = s$). The set of variables and the set of features occurring in ψ are noted respectively as $\mathcal{V}_\psi$ and $\mathcal{F}_\psi$.

The use of variables instead of path references in feature values implies that path equality will be represented as equalities of variables.

For instance, the `Peter` feature term can be described using variables instead of path references as follows:

$$\text{Peter} = X : \mathit{Person}\begin{bmatrix} \mathit{age} & \doteq X_1 : \mathit{28} \\ \mathit{drives} & \doteq X_2 : \mathit{Car}\begin{bmatrix} \mathit{owner} & \doteq X \\ \mathit{complaing} & \doteq X_{21} : \mathit{does\text{-}not\text{-}start} \\ \mathit{gas\text{-}level\text{-}in\text{-}tank} & \doteq X_{22} : \mathit{full} \\ \mathit{gas\text{-}gauge\text{-}reading} & \doteq X_{22} \end{bmatrix} \end{bmatrix}$$

Note that the `Peter` term has a path equality between features `gas-level-in-tank` and `gas-gauge-reading` and another path equality between the `owner` feature and `Peter` (null path).

5.5 Semantics

We have presented the formal syntax for Noos and the set of translation rules from descriptions syntax to feature terms syntax. Next, we will describe the semantical interpretation of feature terms. We have argued our approach to construct terms embodying partial information. In fact, we have said that feature terms are interpreted as partial descriptions.

The semantics of Noos is constructed to capture this notion of feature terms as partial descriptions denoting sets of individuals in a given domain. First of all, we define an interpretation $\mathcal{I}$ over the Noos signature Σ.

Definition 5.5 *(Interpretation)*
We define an interpretation $\mathcal{I}$ over the signature $\Sigma = \{\langle \mathcal{S}, \leq_s \rangle, \langle \mathcal{M}, \leq_m \rangle, \mathcal{F}\}$ as the structure

$$\mathcal{I} = \{\mathbb{S}, \mathbb{F}, \mathbb{M}\}$$

such that:

1. *$\mathbb{S}$ is a non-empty set, called* domain *of $\mathcal{I}$ (or, universe);*
2. *$\mathbb{F}$ is a set of total unary functions $f : \mathbb{S} \mapsto \mathcal{P}(\mathbb{S})$;*
3. *$\mathbb{M}$ is a set of total functions $m : \mathcal{P}(\mathbb{S}) \times \cdots \times \mathcal{P}(\mathbb{S}) \mapsto \mathcal{P}(\mathbb{S})$;*
4. *$\forall s \in \mathcal{S} : [\![s]\!]^{\mathcal{I}} \subseteq \mathbb{S}$ and $\forall s, s' \in \mathcal{S} : s \leq s' \Leftrightarrow [\![s]\!]^{\mathcal{I}} \supseteq [\![s']\!]^{\mathcal{I}}$;*
5. *$\forall f \in \mathcal{F} : [\![f]\!]^{\mathcal{I}} \subseteq \mathbb{F}$;*
6. *$\forall m \in \mathcal{M} : [\![m]\!]^{\mathcal{I}} \subseteq \mathbb{M}$;*
7. *$[\![\perp_s]\!]^{\mathcal{I}} = \mathbb{S}$; and*
8. *$[\![\perp_m]\!]^{\mathcal{I}} = \mathbb{M}$.*

From interpretation $\mathcal{I}$, constant feature terms are interpreted as partial descriptions denoting sets of elements in the domain $\mathbb{S}$ under all possible valuations of its variables $\mathcal{V}_\psi$ in $\mathbb{S}$.

Specifically, the interpretation of a feature term under a specific valuation α of its variables is given as:

Definition 5.6 *(Denotation of a constant feature term under a valuation α)*
Given the interpretation $\mathcal{I}$, the denotation $[\![\psi]\!]^{\mathcal{I},\alpha}$ of a constant feature term ψ, under the valuation $\alpha : \mathcal{V}_\psi \rightarrow \mathbb{S}$ is defined by:

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

$$[\![\psi]\!]^{\mathcal{I},\alpha} = \begin{cases} \{\alpha(X)\} & \textit{when } \alpha(X) \in [\![s]\!]^{\mathcal{I}} \textit{ and } \forall_{i=1,\ldots,n} : [\![f_i]\!]^{\mathcal{I}}(\alpha(X)) \in [\![\Psi_i]\!]^{\mathcal{I},\alpha} \\ \emptyset & \textit{otherwise} \end{cases}$$

expressing that the element assigned to the root variable X (noted $\alpha(X)$) has to belong to the set of elements denoted by its sort $[\![s]\!]^{\mathcal{I}}$, and for each feature f_i the value of the function that denotes $[\![f_i]\!]^{\mathcal{I}}(\alpha(X))$ has to belong to the denotation of the corresponding subterm $[\![\Psi_i]\!]^{\mathcal{I},\alpha}$ under the same valuation.

Since feature values Ψ_i are sets of feature terms, we have to define the interpretation $[\![\psi_1 \cdots \psi_m]\!]^{\mathcal{I},\alpha}$ of sets of feature terms. Sets are interpreted on $\mathcal{P}(\mathbb{S})$ as follows:

Definition 5.7 *(Semantics of sets)*
Given a set $\Psi = \psi_1 \dots \psi_m$, the denotation $[\![\Psi]\!]^{\mathcal{I},\alpha}$ under the valuation α is given by:

$$[\![\psi_1 \dots \psi_m]\!]^{\mathcal{I},\alpha} = \{w \in \mathcal{P}(\mathbb{S}) \mid \exists e_1 \in w \cdots \exists e_m \in w(\forall_{i=1,\dots,m} : e_i \in [\![\psi_i]\!]^{\mathcal{I},\alpha} \text{ and } \forall_{i,j=1,\dots,m} : i \neq j \Rightarrow e_i \neq e_j)\}$$

where for all $\psi_i = X \in \mathcal{V}_\psi$ then $[\![X]\!]^{\mathcal{I},\alpha} = \{\alpha(X)\}$

Finally, constant feature terms are interpreted as the union of domain elements denoted by all the valuations of $\mathcal{I}$ (noted $Val(\mathcal{I})$), as follows:

Definition 5.8 *(Semantics of a constant feature term)*
Given the interpretation $\mathcal{I}$, the denotation $[\![\psi]\!]^{\mathcal{I}}$ of a constant feature term ψ is given by:

$$[\![\psi]\!]^{\mathcal{I}} = \bigcup_{\alpha \in Val(\mathcal{I})} [\![\psi]\!]^{\mathcal{I},\alpha}$$

Using this semantical interpretation of feature terms, it is legitimate to establish an order relation between terms. Given two terms ψ and ψ', we will be interested in determine when $[\![\psi]\!]^{\mathcal{I}} \subset [\![\psi']\!]^{\mathcal{I}}$. In other words, we want to determine when a feature term ψ is more specific (contains more information) than another feature term ψ'.

5.6 Term subsumption

We have just seen that the semantical interpretation of feature terms allows to define an ordering relation between feature descriptions. We call this ordering relation *subsumption*. The intuitive meaning of subsumption is that of *informational ordering*. We say that a feature term ψ subsumes another feature term ψ' (noted $\psi \sqsubseteq \psi'$) when all information in ψ is also contained in ψ'. Formally,

Definition 5.9 *(Subsumption)*
Given two feature terms ψ and ψ', ψ subsumes ψ', $\psi \sqsubseteq \psi'$, if there is a total mapping function $\upsilon : \mathcal{V}_\psi \to \mathcal{V}_{\psi'}$ such that :

1. *$\upsilon(Root(\psi)) = Root(\psi')$,*

and $\forall x \in \mathcal{V}_\psi$

2. *$\tau(x) \leq \tau(\upsilon(x))$,*
3. *for every $f_i \in \mathcal{F}$ such that $x.f_i \doteq \Psi_i$ is defined, then $\upsilon(x).f_i \doteq \Psi'_i$ is also defined, and*
   (a) $\forall \psi_k \in \Psi_i$ either $\exists \psi'_k \in \Psi'_i$ such that $\upsilon(Root(\psi_k)) = Root(\psi'_k)$ or $\exists x' \in \Psi'_i$ such that $\upsilon(Root(\psi_k)) = x'$,

(b) $\forall x \in \Psi_i$ *either* $\exists \psi'_k \in \Psi'_i$ *such that* $\upsilon(x) = Root(\psi'_k)$
or $\exists x' \in \Psi'_i$ *such that* $\upsilon(x) = x'$,

(c) $\forall \psi_k, \psi'_k \in \Psi_i \; (\psi_k \neq \psi'_k \Rightarrow \upsilon(Root(\psi_k)) \neq \upsilon(Root(\psi'_k)))$,

(d) $\forall x, \psi'_k \in \Psi_i \; (\upsilon(x) \neq \upsilon(Root(\psi'_k)))$,

(e) $\forall x, y \in \Psi_i \; (x \neq y \Rightarrow \upsilon(x) \neq \upsilon(y))$.

For instance, given the following two feature terms:

$$\mathrm{T1} = X : Person \begin{bmatrix} name & \doteq X_1 : Name \begin{bmatrix} first \doteq X_{11} : Michael \\ last \doteq X_{12} : Smith \end{bmatrix} \\ lives\text{-}at & \doteq X_2 : Address \big[city \doteq X_{21} : NYCity \big] \\ father & \doteq X_3 : Person \big[name \doteq X_{31} : Name \big[last \doteq X_{12} \big] \big] \end{bmatrix}$$

$$\mathrm{T2} = Y : Person \begin{bmatrix} name & \doteq Y_1 : Name \big[last \doteq Y_{11} : family\text{-}name \big] \\ father & \doteq Y_2 : Person \big[name \doteq Y_{21} : Name \big[last \doteq Y_{11} \big] \big] \end{bmatrix}$$

where `Smith` is a kind of `family-name` (`family-name` $\leq$ `Smith`), we have that T2 $\sqsubseteq$ T1 since T1 contains all the information given in T2 (including the fact that the last name of a person is a `family-name` and is the same that the last name of her father). Moreover, T1 specializes T2 specifying a partial description of her home address, specifying her first name and a specific family-name.

Notice that definition 5.9 of subsumption provides a concrete interpretation of the subsumption between two sets: given two sets of feature terms Ψ, Ψ' we say that $\Psi \sqsubseteq \Psi'$ if the terms provided in Ψ are extended and refined in Ψ'; formally,

Definition 5.10 *(Set subsumption)*
Given two sets of feature terms Ψ, Ψ' *we say that* $\Psi \sqsubseteq \Psi'$ *if for each* $\psi \in \Psi$, *there is a different* $\psi' \in \Psi'$ *such that* $\psi \sqsubseteq \psi'$.

From the definition of subsumption of two feature terms ψ and ψ', it can be easily proved that each subterm ψ_i of ψ also subsumes its correspondent ψ'_i of ψ'. Formally,

Lemma 5.1 *Given two feature terms* ψ *and* ψ', *such that* ψ *subsumes* ψ', *if* $\psi.f_1 \cdots f_n$ *is defined then:*

$$\psi.f_1 \cdots f_n \sqsubseteq \psi'.f_1 \cdots f_n$$

Proof: Let υ *a total mapping function satisfying the subsumption requirements for* $\psi \sqsubseteq \psi'$ *and* $\psi.f_1 \cdots f_n = \Psi$; *since* $\mathcal{V}_\Psi \subseteq \mathcal{V}_\psi$ *we can take the same mapping function restricted to* $\mathcal{V}_\Psi$ *to prove the result.*

The subsumption operation captures the notion of informational ordering. In fact, using subsumption we want to capture the notion of semantical inclusion. When we say that a feature term ψ subsumes another feature term ψ' ($\psi \sqsubseteq \psi'$), we understand that the denotation of ψ ($[\![\psi]\!]^{\mathcal{I}}$) includes the denotation of ψ' ($[\![\psi']\!]^{\mathcal{I}}$). Formally,

Theorem 5.1

$$\psi \sqsubseteq \psi' \Longrightarrow [\![\psi]\!]^{\mathcal{I}} \supseteq [\![\psi']\!]^{\mathcal{I}}$$

Proof:

Since $\psi \sqsubseteq \psi'$, there is a mapping function $\upsilon : \mathcal{V}_{\psi} \rightarrow \mathcal{V}_{\psi'}$ satisfying the subsumption requirements.

Then, for all non empty valuation α' of ψ' we can construct a non empty valuation α of ψ as follows:

$$\alpha(x) = \alpha'(\upsilon(x))$$

Having this valuation, and using Definition 5.9.2, for all $x \in \mathcal{V}_{\psi}$ then $\tau(x) \leq \tau(\upsilon(x))$ is satisfied.

Then, Definition 5.5.4 assures that $[\![\tau(x)]\!]^{\mathcal{I},\alpha} \supseteq [\![\tau(\upsilon(x))]\!]^{\mathcal{I},\alpha}$. In particular, we have that $\alpha(Root(\psi)) \in [\![\tau(Root(\psi))]\!]^{\mathcal{I},\alpha}$.

Next,

1. *if ψ has no features we have that $[\![\psi]\!]^{\mathcal{I}} \supseteq [\![\psi']\!]^{\mathcal{I}}$ and the theorem is proved.*
2. *otherwise for all f such that $\psi.f \doteq \Psi$ is defined, using Definition 5.9.3, we have that $\psi'.f \doteq \Psi'$ is also defined.*

   Finally, since Lemma 5.1 assures that $\Psi \sqsubseteq \Psi'$, by induction we have that $f^{\mathcal{I}}(\alpha(Root(\psi))) \in [\![\Psi]\!]^{\mathcal{I},\alpha}$ and then the theorem is also proved.

From the notion of subsumption we define following additional notion of *equivalence*:

Definition 5.11 *(Equivalence)*
Given two feature terms ψ and ψ', we say that they are syntactic variants if and only if $\psi \sqsubseteq \psi'$ and $\psi' \sqsubseteq \psi$.

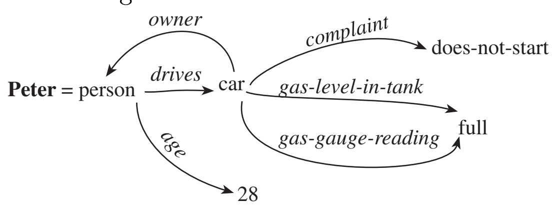
The intuition about the notion of equivalence is that two feature terms are equivalent when contain the same information.

5.7 Representing feature terms as labeled graphs

Feature terms can have an equivalent form of representation as labeled directed graphs [Carpenter, 1992] [Aït-Kaci and Podelski, 1993] [Backofen and Smolka, 1995]. This representation is interesting because it offers an intuitive and visual syntax.

A feature term can be represented as a labeled directed graph that has, for each variable $X : s$, a node q labeled with sort s, and having an arc from q to another node q' labeled by f, written $q \xrightarrow{f} q'$, for each feature f defined in q with feature value q'.

For instance, the graph representation of the feature term of `Peter` from Section 5.4 is the following:



Formally, given the signature Σ, having a set of sort symbols $\mathcal{S}$ and a set of feature symbols $\mathcal{F}$ we define a feature term ψ as a labeled graph as follows:

Definition 5.12 *(Labeled graph representation of feature terms)*
A feature term is a tuple $\psi = < \mathcal{Q}, \bar{q}, \theta, \delta >$ *where:*

- $\mathcal{Q}$ *: is a finite set of nodes rooted at* $\bar{q}$
- $\bar{q}$ *: is the root node*
- $\theta : \mathcal{Q} \rightarrow \mathcal{S}$ *: a total node typing function.*
- $\delta : \mathcal{F} \times \mathcal{Q} \rightarrow 2^{\mathcal{Q}}$ *: is a partial feature value function*

where θ determines the labels of nodes, and $\delta(f, q) = q'$ implies that there is an arc labeled by f from q to another node q'.

In this labeled graph representation path references are defined as follows:

Definition 5.13 *(Paths)*
A path π *is composed of a sequence of features. Let* ϵ *the empty path, we extend* δ *to paths as follows:*

- $\delta(\epsilon, q) = q$
- $\delta(f\pi, q) = \delta(\pi, \delta(f, q))$

That is to say, if $\pi = f_1 \dots f_n$ then $\delta(\pi, q_0) = q_n$ if $q_0 \xrightarrow{f_1} q_1 \xrightarrow{f_2} q_2 \cdots \xrightarrow{f_n} q_n$. The set $\{\delta(\pi, q) | \pi \text{ is a path}\}$ is the set of nodes *reachable* from q by some path. The root of a feature term is a node $\bar{q}$ that satisfies $Q = \{\delta(\pi, \bar{q}) | \pi \text{ is a path}\}$, i.e. the node from which all nodes in Q are reachable.

We say that there is a path equality when two different paths π_1 and π_2 lead from a same node q to the same node q' ($\delta(\pi_1, q) = q' = \delta(\pi_2, q)$).

Labeled graph representation of feature terms has been the basis for developing graphical browsers for the Noos development environment (see Appendix A).

5.8 Understanding feature terms as clauses

Feature terms can be also understood as conjunctions of clauses [Aït-Kaci and Podelski, 1993]. This clausal representation is useful and more usual for understanding learning methods. There are two kinds of atomic clauses: sort clauses ($X : s$) and feature clauses ($f(X, Y)$). A given feature can be represented also as a conjunction of these two kind of atomic clauses.

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

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

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

For instance, the `Peter`'s feature term can be represented as a clause in the following way:

$$\begin{array}{llll} X : \text{Person} & \wedge\ \text{age}(X, 28) & & \\ & \wedge\ \text{drives}(X, Y)\ \wedge\ Y : \text{Car} & \wedge\ \text{owner}(Y, X) & \\ & & \wedge\ \text{complaint}(Y, Z)\ \wedge\ Z : \text{does-not-start} & \\ & & \wedge\ \text{gas-level-in-tank}(Y, W)\ \wedge\ W : \text{full} & \\ & & \wedge\ \text{gas-gauge-reading}(Y, W) & \end{array}$$

5.9 Evaluable feature terms

Noos methods are defined as *evaluable feature terms*. Evaluable feature terms, like constant feature terms, are formalized as recursive records.

Given the signature $\Sigma = \{\langle \mathcal{S}, \leq_s \rangle, \langle \mathcal{M}, \leq_m \rangle, \mathcal{F}\}$ we define evaluable feature terms as follows:

Definition 5.14 *An evaluable feature term ψ is an expression of the form:*

$$\psi \quad ::= [m\ f_1 \doteq \Psi_1 \cdots f_n \doteq \Psi_n]$$

where m is a sort in $\mathcal{M}$, $f_1, \cdots, f_n$ are pairwise distinct features in $\mathcal{F}$, $n \geq 0$, and each Ψ_i is a set of feature terms and variables.

Notice that we said the feature value can be (any) feature term; in particular it can be a constant feature term or an evaluable feature term.

Below, we will describe how Noos built-in methods are formalizated as evaluable feature terms. Then, we will show how new methods are built by refinement.

Built-in methods

The set of Noos built-in methods is formalized as a collection of predefined *evaluable feature terms*. A built-in method is modelled as a recursive extensible record linked to a lambda abstraction. We call "required" features of the evaluable feature term those feature labels defined as the parameters of the lambda abstraction. For instance, the `subtract` built-in method is defined as a recursive extensible record linked to the $subtract_{def}$ lambda abstraction with two named parameters (called `amount` and `minus`) that encodes the usual subtraction operation of two numbers:

$$\begin{array}{lcl} \texttt{subtract} & = & [subtract\ amount \doteq number \quad minus \doteq number] \\ subtract_{def} & = & \lambda(amount\ minus)\ amount - minus \end{array}$$

Defining methods

New methods can be defined by refinement of built-in methods (or other methods). Refinement is modelled as record concatenation. For instance, we can define the method `minus-one` by refinement of the `subtract` method as the following record concatenation:

$$\texttt{minus-one} \ = \ \texttt{subtract} \ll [subtract\ minus \doteq 1]$$

Then, we can define another method by refinement of `minus-one`, binding the remaining `amount` parameter with a specific value obtaining a closed method amenable to be evaluated.

The complete list of built-in methods with their parameter names is described in Appendix D.

5.9.1 Defining methods in features

Closed methods can be evaluated to infer feature values. The evaluation of a method m in a feature f to infer its feature value is indicated with the '#' token using the following syntax:

$$f \doteq m\#$$

The syntax $m\#$ is translated to basic λN syntax using the translation rule $\mathbf{T}$ as follows:

$$\mathbf{T}(m\#) \ = \ m(sel \rightarrow \lambda_{def})!!$$

where λ_{def} is the lambda abstraction linked to the built-in method b such that $b \leq_m m$. That is to say, the evaluation of a method m is translated as the functional application of λ_{def} taking as values for the arguments of λ_{def} the feature values of the features with same name defined in m.

For instance, the following expression

$$[subtract\ amount \doteq 7 \quad minus \doteq 3]\#$$

is translated by the translation rule $\mathbf{T}$ to λN basic syntax as follows

$$\lambda(sel)sel(amount \rightarrow 7)(minus \rightarrow 3)(\lambda(amount\ minus)\ amount - minus)!!$$

that is reduced after a first step to

$$\lambda(amount\ minus)\ amount - minus\ (amount \rightarrow 7)(minus \rightarrow 3)!$$

and eventually yields 4 as result.

We say that a feature is *reduced* when the method incorporated to infer its feature value has been evaluated.

Using the notion of reduced features we introduce the notion of *normal form* for feature terms:

Definition 5.15 *(Normal feature term)*
A feature term ψ is in normal form when all their features ($\mathcal{F}_\psi$) are reduced (there is no feature with a method not yet evaluated).

5.9.2 Semantics of evaluable feature terms

Evaluable feature terms are interpreted, under an interpretation $\mathcal{I} = \langle \mathbb{S}, \mathbb{F}, \mathbb{M} \rangle$, as partial descriptions denoting sets of functions in $\mathbb{M}$. Analogously to constant feature terms, the interpretation of an evaluable feature term is given as the union of functions denoted by all the valuations of $\mathcal{I}$ ($Val(\mathcal{I})$) as follows:

Definition 5.16 *(Semantics of an evaluable feature term)*
Given the interpretation $\mathcal{I}$, the denotation $[\![\psi]\!]^{\mathcal{I}}$ of an evaluable feature term ψ is given by:

$$[\![\psi]\!]^{\mathcal{I}} = \bigcup_{\alpha \in Val(\mathcal{I})} [\![\psi]\!]^{\mathcal{I},\alpha}$$

and for each valuation α in $Val(\mathcal{I})$ its interpretation is given as follows:

Definition 5.17 *(Denotation of an evaluable feature term under a valuation α)*
Given the interpretation $\mathcal{I}$, the denotation $[\![\psi]\!]^{\mathcal{I},\alpha}$ of an evaluable feature term ψ, under a valuation $\alpha : \mathcal{V}_\psi \to \mathbb{M} \cup \mathbb{S}$ is inductively given by:

Let $\psi = X : m[f_1 \doteq \psi_1 \cdots f_n \doteq \psi_n]$

$$[\![\psi]\!]^{\mathcal{I},\alpha} = \begin{cases} \{\alpha(X)\} & \textit{when } \alpha(X) \in [\![m]\!]^{\mathcal{I}} \textit{ and } \forall_{i=1,\ldots,n} : [\![f_i]\!]^{\mathcal{I}}(\alpha(X)) \in [\![\Psi_i]\!]^{\mathcal{I},\alpha} \\ \emptyset & \textit{otherwise} \end{cases}$$

5.10 Preferences

Preferences 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 binary relation $\prec$ defined on S. We demand $\prec$ to satisfy the reflexive and transitive properties. In fact, we are asking $\langle S, \prec \rangle$ to form a pre-order. Formally,

Definition 5.18 *(Preference)*
A preference *is a pair $\langle S, \prec \rangle$, where S is a set of alternatives and $\prec$ is a binary relation over S such that it is reflexive ($a \prec a$) and transitive ($a \prec c$ when $a \prec b$ and $b \prec c$). When $a \prec b$ we say that a is* preferred *to b, and when both $a \prec b$ and $b \prec a$ we say that a and b are* equally preferred[4].

[4] We will note (a_i, a_j) the pairs such that $a_i \prec a_j$.

As we have shown in Section 3.4, there are two kinds of basic operations over preferences in Noos: *preference methods*, that take a set of source elements and an ordering criterion and build a preference (a partially ordered set), and *preference combination methods*, that take two preferences and a combination criterion and build a new preference.

Before to present the formalization of the preference operations we will introduce some preliminary definitions. Then, we will present preference methods, preference combination methods, and finally we show a set of properties about these methods useful for implementing applications in Noos.

Preliminary definitions

Definition 5.19 *(Restriction)*
Given a preference defined by the pair $\langle A, \prec\rangle$ and the set $B \subset A$, we define the preference restricted to the set B as

$$\langle A, \prec\rangle_{|B} = \langle B, \prec \cap \{B \times B\}\rangle$$

Definition 5.20 *(Extension)*
Given a preference defined by the pair $\langle A, \prec\rangle$ and $A \subset B$, we define the preference extension to B as

$$\langle A, \prec\rangle_{extB} = \langle B, \prec \cup \{b \times b\}\rangle$$

Definition 5.21 *(Transitive Closure)*
Given a binary relation $\prec$ defined on a set A we define its transitive closure $\overline{\prec}$ as

$$\overline{\prec} = \{(a_i, a_j) | \exists a_1, \ldots, a_n \in A : a_i \prec a_1 \prec \cdots a_n \prec a_j\}$$

5.10.1 Preference methods

A preference method takes a set of source elements and an ordering criterion and builds a preference. There are several built-in preference methods in Noos. Each preference method implements a different ordering criteria. The complete list of preference methods is provided in Appendix D. Below, we will describe two preference methods as example: `increasing-preference` and `subsumption-preference`.

The `increasing-preference` method can be described as a method that given a feature name f and a set of source elements S builds a preference $\langle S, \prec\rangle$ such that:

$$\prec = \{(s_i, s_j) | s_i.f \doteq v_i,\ s_j.f \doteq v_j,\ v_i > v_j\}$$

The `subsumption-preference` method can be described as a method that given a set of source elements S builds a preference $\langle S, \prec\rangle$ such that:

$$\prec = \{(s_i, s_j) | s_i \sqsubseteq s_j,\ s_j \not\sqsubseteq s_i\}$$

5.10.2 Preference combination methods

A preference combination method takes one or two preferences (created either by preference methods or by other preference combination methods) and builds a new preference combining them in a specific manner. There are five preference combination methods in Noos: `inversion`, `T-intersection`, `C-intersection`, `T-union`, and `C-union`.

The `inversion` method takes a preference $\langle A, \prec \rangle$ and builds a new preference on the same set A inverting the order relations as follows:

Definition 5.22 *(Inversion)*
Given a preference defined by the pair $\langle A, \prec \rangle$ we define its preference inversion as

$$\langle A, \prec \rangle^{-1} = \langle A, \{(a_i, a_j) | (a_j, a_i) \in \prec \} \rangle$$

The other four preference combination methods take two preferences $\langle A, \prec_1 \rangle$ and $\langle B, \prec_2 \rangle$ and build a new preference $\langle S, \prec \rangle$ combining them. Since sets A and B can be different, we will define first two basic combination operations from preferences on the same set of elements. Then, we will explain how combination methods extend these two operations for combining preferences on different sets.

We define two basic operations for combining two preferences $\langle A, \prec_1 \rangle$ and $\langle A, \prec_2 \rangle$, defined on the same set A. The first one builds a new preference mantaining only the pairs $a \prec b$ common to both $\prec_1$ and $\prec_2$ as follows[5]:

Definition 5.23 *(Basic intersection)*
Given two preferences $\langle A, \prec_1 \rangle, \langle A, \prec_2 \rangle$ defined on the same set A we define their intersection preference *as*

$$\langle A, \prec_1 \rangle \cap \langle A, \prec_2 \rangle = \langle A, \prec_1 \cap \prec_2 \rangle$$

The second basic operation builds a new preference gathering all the pairs $a \prec b$ from either $\prec_1$ or $\prec_2$, and performing a transitive closure as follows[6]:

Definition 5.24 *(Basic Transitive Union)*
Given two preferences $\langle A, \prec_1 \rangle, \langle A, \prec_2 \rangle$ defined on the same set A we define their transitive union preference *as*

$$\langle A, \prec_1 \rangle \uplus \langle A, \prec_2 \rangle = \langle A, \overline{\prec_1 \cup \prec_2} \rangle$$

Given two preferences $P1 = \langle A, \prec_1 \rangle$ and $P2 = \langle B, \prec_2 \rangle$ defined on two different sets A and B, we have considered two alternatives in order to be able to use the basic combination operations. The first one is by extending the preferences $P1$ and $P2$ to the union set $A \cup B$. The second alternative considered is to restrict the preferences $P1$ and $P2$ to the intersection set $A \cap B$ (assuming that $A \cap B \neq \emptyset$). Since we have two alternatives and two basic combination operations, we have developed four combination methods.

[5] Note that the intersection of two pre-orders is also a pre-order.

[6] Transitive closure is performed because of the union of two orders is not always an order.

The `T-intersection` method combines two preferences by restricting preferences to the elements of the intersection and, then, performing the transitive union operation on the resulting preferences as follows:

Definition 5.25 *(T-Intersection)*
Given two preferences $\langle A, \prec_1\rangle$, $\langle B, \prec_2\rangle$ we define their t-intersection $\overline{\sqcap}$ using definitions (5.19) and (5.24) as

$$\langle A, \prec_1\rangle \overline{\sqcap} \langle B, \prec_2\rangle = \langle A, \prec_1\rangle_{|A\cap B} \uplus \langle B, \prec_2\rangle_{|A\cap B}$$

The `C-intersection` method combines two preferences by restricting preferences to the elements of the intersection and, then, performing the intersection operation on the resulting preferences as follows:

Definition 5.26 *(C-Intersection)*
Given two preferences $\langle A, \prec_1\rangle$, $\langle B, \prec_2\rangle$ we define their c-intersection $\underline{\sqcap}$ using definitions (5.19) and (5.23) as

$$\langle A, \prec_1\rangle \underline{\sqcap} \langle B, \prec_2\rangle = \langle A, \prec_1\rangle_{|A\cap B} \cap \langle B, \prec_2\rangle_{|A\cap B}$$

The `T-union` method combines two preferences by extending preferences to the elements of the union and, then, performing the transitive union operation on the resulting preferences as follows:

Definition 5.27 *(T-union)*
Given two preferences $\langle A, \prec_1\rangle$, $\langle B, \prec_2\rangle$ we define their t-union $\overline{\uplus}$ using definitions (5.20) and (5.24) as

$$\langle A, \prec_1\rangle \overline{\uplus} \langle B, \prec_2\rangle = \langle A, \prec_1\rangle_{ExtA\cup B} \uplus \langle B, \prec_2\rangle_{ExtA\cup B}$$

Finally, the `C-union` method combines two preferences by extending preferences to the elements of the union and, then, performing the intersection operation on the resulting preferences as follows:

Definition 5.28 *(C-union)*
Given two preferences $\langle A, \prec_1\rangle$, $\langle B, \prec_2\rangle$ we define their c-union $\underline{\uplus}$ using definitions (5.20) and (5.23) as

$$\langle A, \prec_1\rangle \underline{\uplus} \langle B, \prec_2\rangle = \langle A, \prec_1\rangle_{ExtA\cup B} \cap \langle B, \prec_2\rangle_{ExtA\cup B}$$

5.10.3 Higher order preferences

Higher order preferences are preference combination methods that build preferences from preferences over preferences. In the current version of Noos we have developed one higher order preference method called `H-union` (hierarchical union). `H-union` takes two preferences $P1$ and $P2$, and constructs a new preference preserving all the set of order relations specified in $P1$ and adding the subset of order relations from $P2$ that are not in conflict with $P1$.

In a similar way to combination methods, we will first define a basic hierarchical union operation from preferences on the same set of elements. Then, we will explain how `H-union` extends the basic operation for combining preferences on different sets.

Definition 5.29 *(Basic Hierarchical Union)*
Given two preferences $\langle A, \prec_1\rangle, \langle A, \prec_2\rangle$ defined on the same set A we define their basic hierarchical union *preference as*

$$\langle A, \prec_1\rangle \bullet \langle A, \prec_2\rangle = \langle A, \overline{\prec_1 \cup \{(a_i, a_j) \in \prec_2 \mid (a_j, a_i) \notin \overline{\prec_1 \cup \prec_2}\}}\rangle$$

The `H-union` method combines two preferences by extending preferences to the union of sets as follows:

Definition 5.30 *(Hierarchical Union)*
Given two preferences $\langle A, \prec_1\rangle$, $\langle B, \prec_2\rangle$ we define their hierarchical union using definitions (5.20) and (5.29) as

$$\langle A, \prec_1\rangle \bullet \langle B, \prec_2\rangle = \langle A, \prec_1\rangle_{ExtA\cup B} \bullet \langle B, \prec_2\rangle_{ExtA\cup B}$$

5.10.4 Properties

Below we present some properties of preference combination operations that are useful in the development of Noos applications. Particularly, we are interested in properties regarding combinations of preferences—such as commutativity and associativity.

1. The inversion of the inversion of a given preference A yields itself: $(A^{-1})^{-1} = A$
2. Inversion is a morphism with respect to the intersection $(A \cap B)^{-1} = A^{-1} \cap B^{-1}$ and to the union $(A \uplus B)^{-1} = A^{-1} \uplus B^{-1}$.
3. $\uplus$ and $\cap$ operations are associative and commutative.
4. Given two orders A, B (being reflexive, transitive, and antisymmetric) their transitive union $A \uplus B$ is not always an order (may be only a pre-order).
5. Given two orders $\langle A, \prec_1\rangle, \langle A, \prec_2\rangle$, $\langle A, \prec_1\rangle \bullet \langle A, \prec_2\rangle$ has finer granularity than $\langle A, \prec_1\rangle$.
6. Given two orders $\langle A, \prec_1\rangle, \langle A, \prec_2\rangle$ such that $\langle A, \prec_1\rangle$ has finer granularity than $\langle A, \prec_2\rangle$,
$$\langle A, \prec_1\rangle \bullet \langle A, \prec_2\rangle = \langle A, \prec_2\rangle \bullet \langle A, \prec_1\rangle = \langle A, \prec_1\rangle$$
7. Hierarchical union method (5.29) is not commutative.
Example: given $A = \{a, b\}$
$$\langle A, a \prec b\rangle \bullet \langle A, b \prec a\rangle = \langle A, a \prec b\rangle \neq \langle A, b \prec a\rangle = \langle A, b \prec a\rangle \bullet \langle A, a \prec b\rangle$$

8. Hierarchical union method (5.29) is not associative.

   Example: given $A = \{a, b, c\}$

$$\begin{aligned}&(\langle A, a \prec b\rangle \bullet \langle A, b \prec c\rangle) \bullet \langle A, c \prec a\rangle = \langle A, a \prec b \prec c\rangle \bullet \langle A, c \prec a\rangle = \\ &= \langle A, a \prec b \prec c\rangle \neq \langle A, a \prec b\rangle = \\ &= \langle A, a \prec b\rangle \bullet \langle A, b \prec c \prec a\rangle = \langle A, a \prec b\rangle \bullet (\langle A, b \prec c\rangle \bullet \langle A, c \prec a\rangle)\end{aligned}$$

5.11 Perspectives

Perspectives is a mechanism to describe declarative biases for retrieval in the Noos episodic memory. Given a source problem situation expressed by a feature term, and given a syntactic pattern, the `perspective` method constructs a new feature term that is a partial description of the problem describing the relevant aspects of the problem situation. *Perspectives* are constructed in Noos using the `perspective` built-in method (see Section 4.3).

Formally, the signature of syntactic patterns Σ_{ext} is an extension of the signature Σ of feature terms that incorporates a set of feature variables $\mathcal{L}$ as follows:

$$\Sigma_{ext} = \{\langle \mathcal{S}, \leq_s\rangle, \langle \mathcal{M}, \leq_m\rangle, \mathcal{F} \cup \mathcal{L}\}$$

Using this extension of the signature Σ_{ext} and a set $\mathcal{V}$ of variables, we define syntactic patterns as second order feature terms as follows:

Definition 5.31 *A syntactic pattern ω is an expression of the form:*

$$\omega \quad ::= \quad X : s\,[f_1 \doteq \Omega_1 \cdots f_n \doteq \Omega_n]$$

where X is a variable in $\mathcal{V}$, s is a sort in $\mathcal{S}$, $f_1, \cdots, f_n$ are pairwise features in $\mathcal{F} \cup \mathcal{L}$, $n \geq 0$, and each Ω_i is a set of syntactic patterns and variables.

Given the previous definition of syntactic patterns, we define formally a perspective P as follows,

Definition 5.32 (Perspective)
Given a problem case C and a declarative bias defined by means of a syntactic pattern S, a S-perspective *of C is defined as a feature term P such that there is a total bijective function $\beta : \mathcal{V}_P \rightarrow \mathcal{V}_S$, a total mapping function $\delta : \mathcal{V}_S \rightarrow \mathcal{V}_C$, and an instantiation function $\rho : \mathcal{F}_S \rightarrow \mathcal{F}_P$ satisfying:*

1. $\rho(f) = f \qquad \forall f \in \mathcal{F}$
2. $\beta(Root(P)) = Root(S)$, $\delta(Root(S)) = Root(C)$

and $\forall x \in \mathcal{V}_P$

3. $Sort(\beta(x)) \leq Sort(\delta(\beta(x)))$,
4. $Sort(x) = Sort(\delta(\beta(x)))$,
5. *for every $f_i \in \mathcal{F}$ such that $x.f_i \doteq \Psi_i$ is defined, we have that*

(a) $\exists f_j \in \mathcal{F}_S : \rho(f_j) = f_i$,

(b) both $\beta(x).f_j \doteq \Psi'_i$ *and* $\delta(\beta(x)).f_i \doteq \Psi''_i$ *have to be defined,*

and $\forall \psi_k \in \Psi_i$:

(c) $\exists \psi'_k \in \Psi'_i$ *such that* $\beta(Root(\psi_k)) = Root(\psi'_k)$

(d) $\exists \psi''_k \in \Psi''_i$ *such that* $\delta(\beta(Root(\psi_k))) = Root(\psi''_k)$.

Remark that a perspective P is constructed as a partial description of a problem case C. In other words, this implies that $P \sqsubseteq C$. Another important remark is that several perspectives satisfying the definition can be obtained. This implies that the implementation of the perspective mechanism has to provide a way to obtain all of them (for instance, by providing a backtracking based mechanism).

5.12 Descriptive Dynamic Logic

The goal of Descriptive Dynamic Logic (DDL) [Sierra et al., 1996] is to provide a common logical framework to describe and identify the formal characteristics of Multi-Language Architectures (MLA). In this way, DDL can be understood as a formal basis to describe and compare different multi-language architectures.

In general, a MLA allows to build knowledge systems as a set of units with initial local theories written in possibly different languages. Each unit is also usually allowed to have its own intra-unit deductive system. Moreover, the whole knowledge system is equipped with an additional set of deductive rules, called *bridge rules*, to control the information flow among the different units of the knowledge system. Thus, the DDL approach is very useful to model reflective systems based on the use of several units containing local theories (or meta-theories acting upon theories) that influence and/or modify each other.

In order to model knowledge systems using DDL two levels have to be considered: At the first level the *Multi-Language Logical Architecture* (MLA) is defined as the concept representing the most general characteristics of target systems. In this level languages, inference rules, and allowed topologies are described.

At the second level a particular *knowledge system* is represented. Particular theories are built determining a subset of unit identifiers, the languages and inference rules used in each unit, the set of interconnections among units and their corresponding bridge rules, concrete signatures, and finally initial theories for each unit.

For a full description of DDL see [Sierra et al., 1996]. Here we only present some basic definitions of DDL in order to make easier understanding the Noos formalization.

Definition 5.33 *A Multi-Language Logical Architecture is a 4-tuple* $MLA = (L, \Delta, S, T)$, *where:*

1. $L = \{L_j\}_{j \in J}$ *is a set of logical languages,*

2. $\Delta = \{\Delta_{j_1,j_2}\}_{j_1,j_2 \in J}$ *is a set of (instances of) inference rules between pairs of languages, i.e.* $\Delta_{j_1,j_2} \subseteq 2^{L_{j_1}} \times L_{j_2}$. *In particular, when* $j_1 = j_2$, Δ_{j_1,j_2} *denotes a set of inference rules of the corresponding language; otherwise it denotes a set of bridge rules between two different languages,*

3. S *is a finite set of symbols for unit identifiers,*

4. T *is the set of possible topologies. Each topology is determined by a set of directed links between symbols from* S, *i.e.* T *is a subset of* $2^{S \times S}$.

Notice that DDL is focused only on finite languages as it is the usual case in knowledge systems where some limitative rules are imposed on the generation of formulas.

Definition 5.34 *A Multi-Language Knowledge System MKS for a given MLA is a 7-tuple* $MKS = (MLA, U, M_L, M_\Delta, B, M_\Sigma, M_\Omega)$ *where:*

1. *MLA is a Multi-Language Logical Architecture,*

2. U *is a set of unit identifiers, i.e.* U *is a subset of* S,

3. M_L *assigns a language to each unit identifier, i.e.* $M_L \longrightarrow L$,

4. M_Δ *assigns a set of inference rules to each unit identifier, i.e* $M_\Delta : U \longrightarrow \bigcup_{i \in J} 2^{\Delta_{ii}}$ *such that if* $M_L(u) = L_j$, *for some* $j \in J$, *then* $M_\Delta(u) \subseteq \Delta_{jj}$,

5. B *is a mapping that assigns a set of directed bridge rules to pairs of different units, i.e.* $B : U \times U \longrightarrow \bigcup_{i,j \in J} 2^{\Delta_{ij}}$, *in accordance with the allowed topologies in* MLA,

6. M_Σ *assigns a concrete signature to each unit identifier,*

7. M_Ω *assigns a set of formulas (initial local theory) to each unit identifier, i.e.* $M_\Omega : U \longrightarrow \bigcup_{i \in J} 2^{L_i}$ *such that if* $M_L(u) = L_k$ *then* $M_\Omega(u) \subseteq L_k$.

Definition 5.35 *The set* Φ_0 *of atomic formulas of DDL will be defined as the set of "quoted" formulas from the languages* L *in MLA, indexed by the unit identifiers in* U.

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

Definition 5.36 *The set* Π_0 *of atomic programs of DDL is defined as the union of intra-unit inference rules* Π_0^{Intra} *and the inter-unit rules* Π_0^{Inter}

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

being $\Pi_{0_{kk}}^{Intra}$ *the set of all quoted deduction steps allowed according the unit language* $M_L(u_k)$ *and the inference rules determined by* $M_\Delta(u_k)$

and $\Pi_{0_{kl}}^{Inter}$ *the set of all quoted deduction steps allowed according the unit languages* $M_L(u_k)$ *and* $M_L(u_l)$ *and the inference rules determined by* $B(u_k, u_l)$.

Given the set Φ_0 of atomic formulas and the set Π_0 of atomic programs, the set Φ of compound formulas and the set Π of compound programs is constructed following the propositional dynamic logic composition rules defined as:

1. $true \in \Phi$, $false \in \Phi$, $\Phi_0 \subseteq \Phi$,
2. if $\varphi, \psi \in \Phi$ then $\neg\varphi \in \Phi$ and $(\varphi \vee \psi) \in \Phi$,
3. if $\varphi \in \Phi$ and $\alpha \in \Pi$ then $\langle\alpha\rangle\varphi \in \Phi$, denoting the possibility that after the execution of α the formula φ to be true.
4. $\Pi_0 \subseteq \Pi$,
5. if $\alpha, \beta \in \Pi$ then the sequential concatenation $(\alpha; \beta) \in \Pi$,
6. if $\alpha, \beta \in \Pi$ then the indeterministic union $(\alpha \cup \beta) \in \Pi$,
7. if $\alpha \in \Pi$ then the self-iteration $\alpha^* \in \Pi$,
8. if $\varphi \in \Phi$ then $\varphi? \in \Phi$, denoting the program that evaluates whether a given formula φ is true.

$[\alpha]\varphi$ is the usual modal abbreviation for $\neg\langle\alpha\rangle\neg\varphi$. Also $\wedge$, $\rightarrow$ and $\leftrightarrow$ are abbreviations with the standard meaning.

Definition 5.37 *The DDL semantics, following the PDL semantics, is defined relative to a structure* M *of the form* $M = (W, \tau, \rho)$, *where* W *is a set of states,* τ *a mapping* $\tau : \Phi \rightarrow 2^W$ *assigning to each formula* φ *the set of states in which* φ *is true, and* ρ *a mapping* $\rho : \Pi \rightarrow 2^{W \times W}$ *which assigns to each program a set of pairs* (s, t) *representing transitions between states.*

After introducing the basic concepts of DDL, we are ready to formalize the metalevel inference in Noos.

5.13 Modeling Noos inference using DDL

In order to model Noos inference in DDL we have to define the set of unit languages, the set of inta-unit and inter-unit inference rules, the possible topologies, and finally, the inference process as a set of compound programs;

5.13.1 Noos unit languages

Formally, every feature term is represented as a DDL unit. Every unit has a different identifier. There are three kinds of unit languages: *concept languages*, *method languages* and *metalevel languages*. Concept languages pertain to units representing concepts (called concept units). Method languages pertain to units representing methods (called method units) and are an extension of concept languages. Metalevel languages pertain to units representing metalevels (called metalevel units) and are also an extension of concept languages.

Unit languages are built from the set of feature names $\mathcal{F}$ and the set of unit identifiers U. We note U_m the subset of method unit identifiers from U.

Before presenting Noos unit languages we will define a notational equivalence that simplifies their definition:

Definition 5.38 *A feature value c is considered equivalent to the singleton set that contains as element this feature value c.*

$$c \equiv \{c\}$$

Given the definition above, we describe feature values and the set of inference rules for query-methods directly working on sets.

The language L_c of a concept unit c representing a feature term ψ contains formulas describing the feature values pertaining to each feature of ψ and formulas describing the method pertaining to each feature of ψ. Since we have shown in Section 3.3.6 that path references can be also viewed as query-methods, we will represent all the path references as query-methods in order to simplify the DDL model of Noos. Specifically, the set of formulas Φ_c of a given concept unit c is described as:

$$f \in \mathcal{F}, u \in 2^U, m \in U_m : \quad \begin{array}{l} f \doteq u \in \Phi_c \\ f \doteq m\# \in \Phi_c \end{array}$$

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

$$f \in \mathcal{F}; m \in U_m : method(f) \doteq m \in \Phi_\mu$$

$$f \in \mathcal{F}, u \in 2^U : referent(f) \doteq u \in \Phi_\mu$$

The language L_m of a method unit m contains a set of formulas Φ_m describing feature values and feature methods like concept units. Moreover, Φ_m contains a set of formulas describing the result of the method evaluation. These formulas are described as follows:

$$u \in 2^U : result(m) \doteq u \in \Phi_m$$

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

$$f \in \mathcal{F}, c_i \in U, u \in 2^U : c_i.f \doteq u \in \Phi_m$$

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

$$m_i \in U_m, u \in 2^U : result(m_i) \doteq u$$

5.13.2 Inference rules

The elementary inference inference steps in DDL are represented as a collection of inference rules. There are two kinds of inference rules: *intra-unit* inference rules for modeling the inference within a unit, and *inter-unit* inference rules for modeling the communication among the different units. Only methods and metalevel units have intra-unit inference rules. Inter-unit inference rules may connect a unit with any other unit following the Noos topology (see Section 5.13.3).

Intra-Unit Inference Rules

Only methods and metalevel units have intra-unit inference rules. Inference rules in metalevel units select one method for a given feature f from a set of alternative methods S; reflection rules (represented as inter-unit rules) will add this selected method to its referent unit. The inference rule for method selection is the following:

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

Inference rules in methods code the built-in definition of the evaluation of the method. There is one inference rule for each type of built-in method provided by Noos. We have only to translate the previously defined λN lambda abstractions to the DDL syntax. For instance, the inference rule for a `subtract` method m previously defined as

$$subtract_{def} = \lambda(amount\ minus)\ amount - minus$$

is translated to

$$\delta_m^{subs} = \frac{\begin{array}{c} amount \doteq c_1 \\ minus \doteq c_2 \\ \text{“}c' = c_1 - c_2\text{”} \end{array}}{result(m) \doteq c'}$$

where a new formula $result(m) \doteq c'$ is added with the result of the difference between the feature values given in *amount* and *minus*.

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

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

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

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

Inter-Unit Inference Rules

There are four kinds of inter-unit inference rules: *Reification rules*, *Reflection rules*, *Reduction rules*, and *Translation rules*. *Reification rules* specify the representation that a metalevel unit has about its corresponding base-level unit. *Reflection rules* specify the changes that a metalevel unit may perform upon its corresponding base-level unit. *Reduction rules* add to a unit the result of the evaluation of one of its methods. Finally, *Translation rules* specify how formulas may be transported from a unit to another one. The collection of inter-unit inference rules are determined by the allowed topologies in Noos (see Section 5.13.3).

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

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

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

$$\delta^{down}_{\mu c f} = \frac{method(f) \doteq m}{f \doteq \#m}$$

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

$$\delta^{red}_{mcf} = \frac{result(m) \doteq u}{f \doteq u}$$

Translation rules δ^{tquery}_{cmf} add from a unit c to a query-method unit m the formula for a feature f

$$\delta^{tquery}_{cmf} = \frac{f \doteq u}{c.f \doteq u}$$

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

$$\delta^{teval}_{m_1 m_2} = \frac{result(m_1) \doteq u}{result(m_1) \doteq u}$$

5.13.3 Topology

The set of possible topologies in Noos is formed by three kinds of relations among units: *reference relations*, *feature method relations* and *metalevel relations*. The set of reference relations of a unit c with other units c' is determined by the set of formulas $f \doteq c'$ contained in c. The set of feature method relations of a unit c with other method units m is determined by the set of formulas $f \doteq m\#$ contained in c. Metalevel relations are determined by explicit `meta` relations from Noos descriptions. Metalevel relations are exclusive relations: one unit can be the metalevel of only another unit (called *referent*), one unit can only have one metalevel unit, and cycles are forbidden. Note that a metalevel unit can be the referent of another (meta)metalevel unit. Figure 5.5 shows an example of a specific Noos topology.

5.13.4 Programs

Inference in Noos is modelled by means of four kinds of programs: *task programs* formalizing the inference of feature values, *metalevel programs* formalizing the metalevel inference, *query programs* formalizing the inference performed by query methods, and *eval programs* formalizing the inference performed by eval-methods.

The formalization of the Noos inference is presented without taking into account preferences. Then, we will extend the formalization for incorporating preferences.

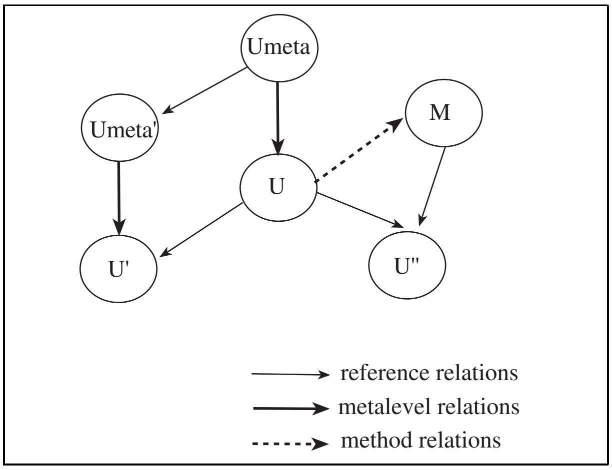


Figure 5.5. An example of an allowed topology.

Task programs

Every unit c has a set Π_c^{Intra} of task programs $\pi_{c.f}$ for features f defined in c. Specifically,

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

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

$$\pi_{c.f} = (\pi^{\mu}_{c.f} \cup true?); \bigcup_{m \in U_m} \left((f \doteq m\#)?; \pi_m; \delta^{red}_{mcf}\right)$$

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

Metalevel inference programs

Metalevel inference $\pi^{\mu}_{c.f}$ for a feature f of a unit c is defined as the following sequential concatenation:

$$\pi^{\mu}_{c.f} = \delta^{up}_c; \pi_{\mu.f}; \delta^{select}_f; \delta^{down}_{\mu cf}$$

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

Evaluation programs

Evaluation of methods in Noos are also formalized as DDL programs having the following scheme:

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

The evaluation program π_m of a method m is composed by the sequence of task programs to infer the values for their required subtasks ($f_1, \cdots, f_n$) followed by the intra-unit inference rule of m that combines the values of subtasks into a final value. For instance, the evaluation program π_m^{sub} of a subtraction method m is the sequence of task programs to compute the operands and the intra-unit rule δ_m^{sub} that combines them.

$$\pi_m^{sub} = \pi_{m.amount}; \pi_{m.minus}; \delta_m^{sub}$$

Query-methods

Query-methods have also their own evaluation programs. The evaluation program of a query-method is the reification of inference in Noos. A query-method m involves the subtask *feature* (for obtaining a feature name f); the subtask *domain* (for obtaining a unit or a set of units s); and the tasks of inferring the feature value of feature f of all units in s. We use s as a shorthand of $\{c_1 \cdots c_n\}$.

The first query-method is *Infer-value* method. The compound program π_m^{iv} for an infer-value method is the following:

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

where

$$R_m^{iv} = ((\mathit{feature} \doteq f) \wedge (\mathit{domain} \doteq \{c_1 \cdots c_n\}))?; \pi_{c_1.f}; \delta_{c_1 m f}^{tquery}; \cdots; \pi_{c_n.f}; \delta_{c_n m f}^{tquery}$$

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

The evaluation program π_m^{ev} of an *Exists-value* method m determines if any solution to a task f for a set of units in s exists, and returns a boolean value accordingly:

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

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

The evaluation program π_m^{kv} of a *Known-value* method m determines if the solution to a task f for a set of units in s have already been computed, and returns a boolean value accordingly:

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

where

$$\begin{aligned} R_m^{kv} &= ((\mathit{feature} \doteq f) \wedge (\mathit{domain} \doteq \{c_1 \cdots c_n\}))?; \\ &\quad \underline{\text{if}}\ (\delta_{mc_1 f}^{tquery}; \cdots; \delta_{mc_n f}^{tquery})?\ \underline{\text{then}}\ \delta_m^{true}\ \underline{\text{else}}\ \delta_m^{false} \end{aligned}$$

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

Finally, evaluation program π_m^{av} of an *All-values* method m determines the set of all inferrable values to a task f for a set of units in s:

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

where

$$\begin{aligned} R_m^{av} &= ((\mathit{feature} \doteq f) \wedge (\mathit{domain} \doteq \{c_1 \cdots c_n\}))?; \\ &\quad (\pi_{c_1.f}; \delta_{mc_1 f}^{tquery})^c; \cdots; (\pi_{c_n.f}; \delta_{mc_n f}^{tquery})^c \end{aligned}$$

and where α^c represents the closure of program α—that is, this program will lead to a state in which no different state is reachable by another application of program α. Specifically, a closure $(\pi_{c_i.f}; \delta_{mc_i f}^{tquery})^c$ produces all possible values of task program $\pi_{c_i.f}$. Next δ_m^{av} rule puts together all the values.

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

Eval-methods

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

The first eval-method is *Noos-eval*. The evaluation program π_m^{ne} of a *Noos-eval* method m is the following:

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

where

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

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

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

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

The evaluation program of a *Known-eval* method m is analogous to the previous evaluation program for determining that methods have already been evaluated:

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

where

$$R_m^{ke} = (methods \doteq \{m_1 \cdots m_n\})?; \underline{\text{if}} \ (\delta^{ne}_{m_1 m}; \cdots; \delta^{ne}_{m_n m})? \ \underline{\text{then}} \ \delta_m^{true} \ \underline{\text{else}} \ \delta_m^{false}$$

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

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

where

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

and δ_m^{ae} rule puts together all the values.

Inference in Noos 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*. A path reference `(>> f of d)` will start the task program $\pi_{d.f}$. An eval expression `(noos-eval m)` will start the eval program π_m.

5.13.5 Adding preferences

Several alternative solutions can be yielded in solving a problem task `T`, since several methods can be defined for solving a subtask of `T`. The DDL-based formalism for modeling Noos inference presented in previous section yields a solution for a problem task `T` in an indeterministic way. Preferences are used in Noos for specifying a preference order on the set of alternative methods defined for solving a task. This preference order between methods implicitly define a preference order on the values inferred by methods. Thus, preferences define a preference order on solution values for a problem task T.

Preferences can be added to the previous presented formalism as a postcondition constraining the results yielded for a problem task. Specifically, given a problem task $f(d)$ we can define the task program $\pi^*_{d.f}$ taking into account the preference information as the following contatenation:

$$\pi^*_{d.f} = \pi_{d.f}; (Maximal\_Solution)?$$

where first $\pi_{d.f}$ engages the indeterministic task program for inferring a solution value; and $Maximal\_Solution$ is a conjunction of DDL formulas that ensures that the solution is maximal with respect to preferences.

The DDL expression for $Maximal\_Solution$ is a conjunction of formulas satisfying the following definition:

Definition 5.39 *(Maximal solution)*
Given the set of achieved subtasks $t_1, t_2 \cdots t_n$, that form the task decomposition of a problem task T, given the set of partial orders $\prec_1, \cdots \prec_n$ over the alternative methods for these subtasks, and given the set of methods $m_1, m_2 \cdots m_n$ engaged respectively to these subtasks, a solution of T is maximal *if there is no other combination of methods $m'_1 \prec_1 m_1$, $m'_2 \prec_2 m_2, \cdots m'_n \prec_n m_n$ (where at least one $m'_i \neq m_i$) that achieves a solution for T.*

5.14 Summary

This chapter presented the formal description of the Noos language. We presented the Noos formal syntax based on feature terms, its semantics, and the formal model of the Noos inference process.

We used λN calculus to provide a syntax for Noos feature terms. Moreover, λN calculus capabilities for modeling extensible knowledge are used for modeling the refinemement mechanism of Noos. λN calculus lexical scoping is used for modeling path references and path equality.

We adopted a related approach to the semantical interpretation of ψ-terms in order to provide a semantical interpretation of Noos feature terms. Following the *ψ-term* formalism, feature terms are interpreted as partial descriptions. This semantical interpretation of feature terms brings an ordering relation among them. We call this ordering relation *subsumption*.

ψ-terms allows three equivalent representations: terms, clauses, and graphs. We presented a graph representation of feature terms. The graph representation is the basis for developing graphical browsers for the Noos development environment (see Appendix A). We also presented the clausal representation. clausal representation is useful for comparing Noos learning methods with other existing learning methods.

We follow the work on *ψ-terms* for providing a formalism for Noos. Nevertheless, other formalisms such as description logics [Nebel, 1990] (also known as terminological logics) are also close to ψ-*terms*. Examples of languages based on description logics are LOOM [MacGregor, 1991] and CLASSIC [Brachman et al., 1991].

We used Descriptive Dynamic Logic to describe the inference process involved in solving a specific problem task. The DDL model of Noos is defined by means of a collection of units with three kinds of unit languages: concept languages, method languages and metalevel languages. Every feature term is represented as a DDL unit. The elementary inference inference steps in Noos are represented as a collection of inference rules. There are two kinds of inference rules: intra-unit inference rules for modeling the inference within a unit, and inter-unit inference rules for modeling the communication among the different units. Only methods and metalevel units have intra-unit inference rules. Inter-unit inference rules may connect a unit with any other unit following the Noos topology. Then, combining the inference rules, inference in Noos is modeled by means of four kinds of programs: task programs formalizing the inference of feature values, metalevel programs formalizing the metalevel inference, query programs formalizing the inference performed by query methods, and eval programs formalizing the inference performed by eval-methods.

Finally, we defined formally two specific elements of Noos: preferences and perspectives. Preferences are a declarative mechanism for decision making about sets of alternatives. Reasoning with preferences is modeled by partially ordered sets with a set of operations for constructing new preferences and combining them.

Perspectives are a mechanism to describe declarative biases for retrieval in the Noos episodic memory. Using feature terms, perspectives are formalized as second order feature terms that denote sets of terms.

Chapter 6

Applications

The purpose of this chapter is to provide a set of examples of how diverse applications have been developed using Noos by several persons at the IIIA.

All these applications are described in detail in other publications. The goal of this chapter is to describe their main characteristics. Following this purpose, and after a brief introduction of the task that the application solves, we will focus on three aspects of the applications: how the domain knowledge required for problem solving methods is modeled using Noos representation capabilities, which problem solving methods are developed to perform the task, and which learning methods are incorporated for solving the task.

In this chapter we will present six applications developed using Noos: CHROMA, SPIN, SHAM, GYMEL, *Saxex*, and NoosWeb.

CHROMA is a system for recommending a plan for the purification of proteins from tissues and cultures using chromatographic techniques developed by Eva Armengol [Armengol and Plaza, 1994] [Armengol, 1997].

SPIN is a sponge identification system for a class of marine sponge species (the family of *Geodiidae*) also developed by Eva Armengol [Armengol, 1997].

SHAM is a tool to help a non expert musician to harmonize melodies using background musical knowledge developed by Martí Cabré [Cabré, 1996].

GYMEL is also a system for harmonization of melodies. Nevertheless, GYMEL harmonize melodies using a case-based reasoning approach. GYMEL has been developed by Jordi Sabater [Sabater, 1997].

Saxex is a case-based reasoning system for generating expressive performances of melodies based on examples of human performances that are represented as structured cases. I have developed *Saxex* in the context of my M.Sc.Thesis in Computer Music [Arcos, 1996] [Arcos et al., 1997b].

Finally, NoosWeb is a WWW interface to Noos applications supporting the same interaction capabilities provided in the Noos window-based graphical interface developed for Apple computers. NoosWeb is not exactly an application such as other applications presented in this chapter. We have included NoosWeb for presenting the accessibility facilities of Noos and for describing the set of small improvements performed in Noos to support remote calls. NoosWeb has been developed by Francisco Martín [Martín, 1996].

6.1 CHROMA

Noos has been used to implement CHROMA [Armengol and Plaza, 1994] [Armengol, 1997], a system for recommending a plan for the purification of proteins from tissues and cultures using chromatographic techniques. Purification is an essential process in the analysis of the properties of molecules from biological origin and widely used in industry and research. Proteins are a type of biological macromolecules that are purified by a sequence of laboratory operations. The most used operations for protein purification are chromatographic techniques. Purification plans of CHROMA incorporate different chromatographic techniques such as:

- *Ion-exchange*: process based on the Coulomb's law,
- *Hydrophobic Interaction*: process based on the Van der Vaals law,
- *Gel Filtration*: process that separates the molecules according its size, and
- *Affinity*: process that exploits the existence of specific unions between certain types of molecules.

Moreover, CHROMA incorporates different techniques such as *precipitation* and *clarification*, previous to the chromatographic process.

A plan to purify a molecule can be composed by several steps involving a chromatographic technique, that can be different, in each step. There is no unique way to purify a given protein. To choose an adequate purification plan involves reasoning about different aspects such as, for instance, the protein to purify, the sample origin (culture, tissue, etc), and the future use of the purified molecule. Protein purification requires the experience of an expert. Usually, a human expert first carries out a focused search in the literature in order to obtain a set of purification plans used in "similar" problems. Then, the expert analyzes the set of collected precedents and chooses the most appropriate.

The goal design of CHROMA was to build a system that, using a memory of purificated cases, was capable to find precedent cases useful for solving new experiments in an expert-comparable way. CHROMA learns from experience using two learning methods: CBR learning and induction. Four problem solving methods have been developed for recommending purification plans. One of them is a classification method that uses the induced knowledge. Moreover, a metalevel method is able to decide, for a particular problem, which problem solving method is more likely to succeed. The reflective capabilities of Noos allow CHROMA to analyze and decompose problem solving and learning methods in a uniform way, and also to combine them in a simple and efficient way.

We will give here a brief description of CHROMA's components. The reader may consult [Armengol and Plaza, 1994] and [Armengol, 1997] for a more detailed description of CHROMA.

Figure 6.1. A Noos browser of an experiment from CHROMA's case-base. An experiment is composed of two features: the sample from which the protein has to be extracted and the purification plan. In this experiment purification plan is formed by three steps `precipitation`, `ion-exchange`, and `affinity`.

6.1.1 Modeling domain knowledge

The domain ontology of CHROMA is composed of concepts such as experiments, samples, purification plans, proteins, species, tissues, and chromatographic techniques.

Experiments have two features: the `description` feature, embodying a description of the protein to be purificated and of the sample from which the protein has to be extracted, and the `purification` feature, embodying the purification plan to be performed (see Figure 6.1). A `description` is a feature term with two features: the `protein` to purify and the `sample`. A `sample` is described, in turn, by two features: the `species` where the sample comes from, and the `source` of the sample (such as an animal or vegetal tissue, or a culture).

Purification plans are composed of a variable number of chromatographic steps represented as features of the plan (called `step1`, `step2`, etc) embodying the step. Each step has two main features: the `name` feature, containing a specific chromatographic technique (`Affinity`, `ion-exchange`, etc), and either the `reagent` or the `resin` feature containing the substance used to purify the protein. Other complementary features such as the `PH` feature are also defined.

CHROMA has available an episodic memory of about one hundred solved purifications. Moreover, the system has been tested using twenty-five new problems.

6.1.2 Solving the purification task

The main task of the CHROMA application is the *purification* task. Given a new experiment containing only the sample, the goal of the *purification* task is to determine a purification plan using domain knowledge and the episodic memory of purificated cases. The purification task uses four methods for recommending purification plans: `equal-sample`, `analogy-by-determination`, `classify-by-prototype`, and `default-plan`

Equal-Sample method

The `equal-sample` method detects if there is an experiment in the episodic memory having the same sample as our current experiment.

Specifically, the `equal-sample` method is a CBR method decomposed in two subtasks: `retrieve` and `reuse` subtasks.

The `retrieve` subtask is achieved by a method defined by refinement of the `retrieve-by-pattern` built-in method. The retrieval method takes the sample of the current experiment problem as a pattern to perform a search for precedents into the episodic memory. The `reuse` task is achieved by reinstantiating the purification plan given in the precedent found to the current problem.

The `equal-sample` method is useful to solve routine purifications with commonly occurring samples and proteins assuring a correct solution for these cases.

Analogy-by-determination method

The `analogy-by-determination` method is a case-based method based on analogical *determinations* [Russell, 1990]. Determinations are functional dependencies used as justifications in analogical reasoning (see an example in Section 4.2).

The `analogy-by-determination` method is a method decomposed in three subtasks: `retrieve`, `select`, and `reuse` subtasks.

The `retrieve` task is achieved by a retrieval method that searches for experiments from the episodic memory purifying the same protein than the current experiment problem—that is to say, stating that the purification plan depends on the protein.

When the retrieval method finds several precedents, the method defined for the `select` task ranks the precedents according to domain specific criteria such as similarity of species or source. When domain criteria are not sufficient to determine a most preferred precedent, the precedents are presented to the user who must choose one of them.

Finally, the `reuse` subtask is achieved by reinstantiating the purification plan given in the most preferred precedent, according to the inference performed by the `select` subtask, to the current problem.

Classify-by-prototype method

The `classify-by-prototype` method is a classification method. This method is decomposed in four subtasks: `obtain-classes`, `plausible-classes`, `select`, and `reuse`.

The `obtain-classes` task is achieved by a method that obtains the set of solution classes in which a new experiment can be classified. This set of solution classes is generated using inductive learning by the `induce-prototype` method (see next method). Solution classes are defined as feature terms embodying a purification plan and a prototype that is a partial description of an experiment description generalizing the set of experiments that has been solved using this purification plan.

The `plausible-classes` task selects, using a method based on subsumption, the subset of solution classes which description subsumes the new experiment.

When more than one solution class is chosen, the `select` task has defined an user-interface method that presents their associated purification plans to the user and asks to the user to select the best of them (the user required to be himself the responsible instead of automating further the process).

As in all CHROMA methods, the `reuse` subtask is achieved by reinstantiating the purification plan given in the most preferred precedent, according to the inference performed by the `select` subtask, to the current problem.

Induce-prototype method

The `induce-prototype` method is an inductive method based on antiunification (see Section 4.6). The goal of the `induce-prototype` method is to construct a set of purification prototypes of experiments that share the same purification plan. These set of prototypes are used by the `classify-by-prototype` method.

The `induce-prototype` method is decomposed in three subtasks: `build-partitions`, `select-representative-sets`, and `generate-prototype`.

The first subtask divides the base of cases into sets containing experiments purified following the same purification plan.

Because of some of the formed sets may have few elements and it is not desirable to make induction with these small sets, the `select-representative-sets` task, defined by a filter method, rejects those sets having a number of elements lower than a threshold.

Finally, the `generate-prototype` subtask constructs, using methods based on antiunification such as INDIE (a bottom-up induction method) or DISC (a top-down induction method) [Armengol, 1997], a purification prototype for each representative set.

Default-plan method

The `default-plan` method is a domain method based on a statistical analysis of purification experiments. This method is used when there is no experiment in the episodic memory purifying the protein we are interested in and other methods are failed.

The `default-plan` method is based on the observation that when an expert doesn't find similar precedents in the literature for solving a given problem, the expert uses generally valid default plans. These default plans are less efficient but nonetheless appropriate for purification. The `default-plan` method captures this observation recommending a plan obtained from a statistical analysis of purification experiments.

Configuration of methods

CHROMA is provided by a metalevel reasoning method to decide, on a case-by-case basis, the applicability of the CHROMA methods and the order in which they are attempted. The metalevel method analyzes the current problem in order to determine the subset of applicable methods and an order for attempting methods. An empirical assessment has shown that this approach is better for this domain than the usual fixed sequence of methods attempted until one succeeds.

The implementation of the CHROMA metalevel reasoning method illustrates the powerful capabilities of Noos to implement, combine and experimentally evaluate the quality and efficiency of methods and method-combinations, and tailor the system to the task requirements in an application domain.

6.2 SPIN

SPIN [Armengol, 1997] is another system developed using Noos at the IIIA. SPIN is a sponge identification system for a class of marine sponge species (the family of *Geodiidae*). SPIN currently integrates a bottom-up induction method, a top-down induction method, a CBR method based on an entropy measure, and a method that combines lazy induction and CBR.

The identification of marine sponge specimens is a specially complex task that requires an expert due to the genetic diversity and the morphological plasticity of marine sponges. Moreover, in some sponge phylums, such as the Porifera phylum, is not clear how the different taxa are characterized.

SPIN knowledge base has been constructed from a subset of the specimens used to test the SPONGIA system [Domingo, 1995], a knowledge based system implemented at our Institute using MILORD-II [Puyol-Gruart, 1995]. An important remark is that a specific sponge specimen is described in SPIN using only the set of features used by SPONGIA in its identification.

A specimen can be identified as belonging to five different taxonomic levels (*class*, *order*, *family*, *genus*, and *species*). Currently, SPIN knowledge base contains only specimens from the *Geodiidae* family. This implies that SPIN can only identify specimens at *genus* and *species* levels. Nevertheless, incorporating specimens from other taxa, SPIN could perform identification at the five taxonomic levels.

Analogously to the description of CHROMA, we will give here a brief description of SPIN components. The reader can consult [Armengol, 1997] for a more detailed description.

Figure 6.2. A Noos browser of a sponge problem from SPIN's case-base. A sponge problem is composed of two features: the `description` of a sponge specimen and its classification into the five taxonomic levels.

6.2.1 Modeling domain knowledge

The domain ontology of SPIN is composed of concepts such as specimen descriptions, skeleton characteristics, and taxa.

Cases are described by refinement of a `sponge-problem`. A `sponge-problem` is a feature term holding two features: `description` and `solution` features (see Figure 6.2). The feature value of a `description` feature is a feature term describing the specific information of a sponge specimen. The feature value of a `solution` feature embodies a feature term describing the classification of the sponge specimen into the five taxonomic levels.

6.2.2 Solving the identification task

The main task of the SPIN application is the *identification* task. Given a new sponge specimen, the goal of the *identification* task is to determine the taxa (genus and species) to which this new sponge specimen belongs[1]. There are two alternative ways to proceed in order to identify specimens: (1) analyzing whether a new specimen has the features characteristic of some taxa, or (2) looking for similar specimens into the episodic memory and classifying the new specimen according to the taxa of the most similar precedents.

[1]They are all of the Geodiidae family, so there is no need of identification at family or higher levels

The identification task can be achieved using three different methods: `identify-by-subsumption`, `CRASS`, and `LID`. Moreover, the `identify-by-subsumption` method may use domain knowledge built by two inductive learning methods: either by INDIE or DISC.

Identify-by-subsumption method

The `identify-by-subsumption` method is a classification method based on domain knowledge acquired by means of an inductive learning method. The goal of `identify-by-subsumption` method is to classify a new sponge specimen from a set of concept descriptions of the taxa in which the new specimen can be classified. Concept descriptions are acquired using either INDIE or DISC. The decision of which inductive method to use is made dynamically by the user.

The `identify-by-subsumption` method is a method decomposed in four subtasks in a similar way of `classify-by-prototype` method developed in CHROMA.

CRASS

CRASS is a domain independent case-based method that uses an entropy-based assessment of similitude importance [Plaza et al., 1996b]. The goal of CRASS is to classify a new case, from a set of solution classes and a set of precedent cases, estimating its similarity to precedent cases.

The CRASS method is decomposed in two subtasks: `build-similarity--terms` and `select` tasks.

The goal of the `build-similarity-terms` terms subtask is to build a set of *similarity terms* from the new case N and each precedent case P_i. A similarity term is a partial description, built using antiunification, containing the commonalities between N and P_i.

Once the set of similitude terms are generated, the goal of the `select` case is to choose the precedent most similar to the new case using similitude terms. Specifically, the `select` subtask is performed by a method decomposed, in turn, in two subtasks: `entropy-assessment` and `weight` subtasks. The first `entropy-assessment` subtask estimates the importance of a similitude term S by measuring the entropy of the set of precedents subsuming S with respect to solution classes. This estimation is performed by a method based on Shannon entropy. The second `weight` subtask uses an aggregation-based method for weighting the entropy measure of each solution class with respect to the cardinality of the set of precedents belonging to this class. The solution class selected is that of with higher weight.

LID

LID is a domain independent method that combines top-down lazy induction and case-based reasoning. The goal of LID is also to classify a new case, from a set of solution classes and a set of precedent cases.

The LID method is decomposed in two subtasks: `build-description` and `identify` tasks.

The goal of the first `build-description` task is to construct a partial description of the new problem. The partial description is constructed incrementally, from an empty description, by adding one feature (the most discriminating one) to the description. The task is performed by a method that combines antiunification and a discriminant measure of features' relevance based on López de Mántaras distance [López de Mántaras, 1991].

The goal of the `identify` task is to discriminate the new problem, using the partial description constructed by the `build-description` task, with respect to precedent cases. The method developed to perform such task is based on subsumption and on Shannon entropy.

`Build-description` and `identify` subtasks are reiterated until the partial description constructed by the `build-description` task discriminates the new problem with respect to all precedent cases and it classifies it in a solution class, or there is no more discriminant features. In the first option, LID yields that solution class. In the second option, LID yields a set of ranked possible solution classes.

6.3 SHAM

SHAM [Cabré, 1996] is also a system developed using Noos. The main goal of SHAM was to develop a tool to help a non expert musician to harmonize melodies using background musical knowledge.

Musical knowledge is described by means of (1) PSMs that characterize local situations (for instance notes with specific weight in a metre), (2) PSMs that propose chords for these local situations, and (3) PSMs that combine the alternatives constructing a complete chord sequence.

Currently, SHAM takes a MIDI file[2] containing a melody line, translates the information to Noos descriptions, and starts an inference cycle of Noos obtaining a chord sequence and a bass line described in Noos. Finally, SHAM generates a new MIDI file containing the melody line plus the chord sequence and the bass line[3].

The user is not required to know neither Noos nor the internal musical representation of the system. SHAM is provided with a window-based interface that allows to choose an input MIDI file, specify musical information not provided in the MIDI file such as the key and the metre, and choose a set of alternative parameters stating the harmonization style. These set of harmonization parameters determine the set of used chords (e.g. not taking into account sevenths in chords and the maximum number of chord inversions) and the degree of complexity of the final chord sequence.

[2]MIDI files are the standard format for representing musical scores and can be generated by all computer music editors.

[3]The reader can visit our web site at <http://www.iiia.csic.es/Projects/music/> for sound examples.

Figure 6.3. A Noos browser of the song 'El noi de la mare' from SHAM. A problem to be solved in SHAM is described as a work with two features: `parts` describing the hierarchical decomposition of a musical piece in terms of bars and notes, and `harm-type` describing the user preferences for harmonization.

An interesting feature of SHAM is that different runs may result in different harmonizations depending on that several parameters and on a random component giving, therefore, the possibility to explore and combine different results. SHAM is being applied to harmonize catalan folk songs and sometimes shows a good degree of creativity.

6.3.1 Modeling musical knowledge

The domain ontology of SHAM is composed of concepts analyzing musical pieces in a hierarchical way (see Figure 6.3): a piece is represented as a `work` decomposed in several `parts`. Each `part` is able to have a different key. Moreover, a `part` is decomposed in several `bars`. Finally, every `bar` holds a set of `notes` and a set of `chords`.

Specifically, a `work` is a feature term with five features: `parts`, holding the set of parts of the work; `harm-type`, holding the harmonization parameters chosen by the user; `selected-chords`, holding the subset of applicable chords following the hamonization style specified in `harm-type`; `chord-sequence`, holding the final chord sequence inferred by SHAM; and `bass-line`, holding the final base line inferred by SHAM.

Feature values of features `parts` and `harm-type` are determined by using an input graphical interface. Feature `selected-chords` is defined using a filtering method that infers a subset of chords from the set of all defined chords and the set of user preferences defined in feature `harm-type`. Features `chord-sequence` and `bass-line` specify a possible harmonization of the piece and are inferred using problem solving methods defined for the harmonization task (see next subsection).

A `part` is a feature term with five features: `from-work`, that is a reference to the work it belongs; `order`, holding a reference number identifying the part; `key`, holding the key of the part; `mode`, holding the key mode (`major` or `minor`[4]); and `bars`, holding the set of bars of the part.

A `bar` is a feature term with two kinds of features: features given as problem data (such as `from-part`, `order`, `metre`, and `notes`) and features inferred, using background musical knowledge, while selecting the chord sequence (such as `vertical-choose`, `horizontal-choose`, `final-chords`, and `final-bass`). Feature `from-part` is a reference to the part it belongs. Feature `order` holds a reference number identifying the bar. Feature `metre` holds the metre of the bar. Feature `notes` holds the set of notes contained in the bar. Inferred features will be described in next section.

A `note` is a feature term with six features: `from-bar`, that is a reference to the bar it belongs; `pitch`, holding the pitch relative to the current key (e.g. pitch `C` in key `C` is represented as `p1`); `onset`, holding the delay from the start of the bar to the start of the note; `duration`, holding the note's duration; `octave`, holding the note's octave (only used for MIDI interface); and `chord-note?`, stating if the note is important enough to have a chord. Feature `chord-note?` is defined by means of a method that estimates the importance of the note from its `duration` and its `onset`.

A `chord` is a feature term with five features: `from-bar`, that is a reference to the bar it belongs; `name`, holding the kind of chord (e.g. `Imaj7`); `onset`, holding the delay from the start of the bar to the start of the chord; `duration`, holding the chord's duration; and `weight`, stating the appropriateness of the chord in the bar. Feature `weight` is defined by means of a method that estimates the suitability of the chord taking into account the notes of the bar it belongs. SHAM deals with more than one hundred different chords.

6.3.2 Solving the harmonization task

The main task of SHAM is the *harmonization* task. Given the melody of a musical piece, the goal of the *harmonization* task is to find a sequence of chords and a bass line for this piece. The chord sequence is built taking into account user

[4] currently SHAM supports only major modes since there are no harmonization methods dealing with minor modes. Nevertheless, adding new harmonization methods minor modes could be harmonizated.

preferences and background musical knowledge expressed as methods. The bass line is built using background musical knowledge and according to the chord sequence.

The selection of a chord sequence is more complex than the selection of an appropriate bass line. Here we only sketch the problem solving method developed by selecting chord sequences. The reader can consult [Cabré, 1996] for a detailed description of the harmonization process in SHAM.

The main reasoning in selecting chords is performed in the context of a bar. SHAM can choose one chord covering a whole bar, two chords covering each of them half a bar, a chord covering half a bar (the beginning or the end), or a rest (a bar without chords). The inference process engaged for each bar is decomposed in four subtasks: (1) first the set of chords available according to the user preference is taken from the work feature `selected-chords`, (2) then task `vertical-choose` is engaged in order to filter the subset of feasible chords according to the notes of the bar, (3) next task `horizontal-choose` estimates the appropriateness of chords according to neighbor bars, (4) finally task `final-chords` selects a chord (or a set of chords) for the bar taking into account the weight of chords and a random factor.

6.4 GYMEL

GYMEL [Sabater, 1997] is also a system developed in Noos for harmonization of melodies. The main goal of GYMEL was to develop a tool to help a non expert musician to harmonize melodies using a case-based reasoning approach. Moreover, GYMEL incorporates background musical knowledge for solving isolated situations not covered by existing cases.

The approach taken in GYMEL is quite different to SHAM's approach. SHAM is based on a hierarchical decomposition of a piece, given as input, from which the selection of possible chords is performed. On the other side, GYMEL starts from a melody line (a sequence of notes) constructing an analysis structure over that melody line that is the basis for retrieval of similar precedents.

GYMEL, as SHAM does, takes a MIDI file containing a melody line, translates the information to Noos descriptions, starts an inference cycle of Noos obtaining a chord sequence described in Noos, and finally generates a new MIDI file containing the melody line plus the chord sequence.

GYMEL suggests different harmonizations for a given musical phrase. The user can select some of them to be incorporated into the episodic memory. These harmonizations will then be used in solving new phrases.

6.4.1 Modeling musical knowledge

The domain ontology of GYMEL is composed of concepts such as phrases, keys, notes, chords, and nodes.

A musical piece is described as a `phrase`. A `phrase` is a feature term with six features: `key`, holding the key of the piece; `mode`, holding the key mode

Figure 6.4. A Noos browser of a musical phrase from GYMEL's case-base. A Case is represented as a feature term with six features: `key`, `mode`, and `metre` describe general knowledge of the phrase; `melody` holds a sequence of notes; `structure` holds a sequence of nodes; and `harmony` holds a sequence of chords.

(`major` or `minor`); `metre`, holding the metre of the piece; `melody`, holding the melody line as a sequence of notes; `structure`, holding the analysis structure of the piece as a sequence of nodes; and `harmony`, holding a sequence of chords. Figure 6.4 shows a browser of a phrase structure.

A `note` is a feature term with six features: `value`, holding the corresponding MIDI number of the note (used by the MIDI interface); `name`, holding the name of the note (e.g. `C4`); `pitch`, holding the pitch relative to the current key (as same as SHAM); `dur-rel`, holding the time distance from the beginning of the phrase to the note; `duration`, holding the note's duration; and `next`, holding a reference to the following note in the phrase.

A `chord` is a feature term with four features: `kind`, holding the kind of the chord (e.g. `IIminor7`); `dur-rel`, holding the time distance from the beginning of the phrase to the chord; `duration`, holding the chord's duration; and `next`, holding a reference to the following chord in the phrase.

The analysis structure is composed of a sequence of `nodes` grouping sets of consecutive notes in the phrase. Each note will belong to a unique node, and for each node one chord will be selected. A `node` is a feature term with six features: `notes`, holding a group of consecutive notes; `main-note`, holding the most important note of the group; `metre`, holding a reference to the metre of the phrase; `chord`, holding the chord to be played at same time than the group of notes; `next`, holding a reference to the following node; and `prev`, holding a reference to the previous node.

6.4.2 Solving the harmonization task

The main task of GYMEL is the *harmonization* task. Given the melody of a musical phrase, the goal of the *harmonization* task is to find a sequence of chords for this phrase. The *harmonization* task is performed by a method decomposed in two subtasks: `build-analysis-structure` and `propose-harmony`.

The goal of task `build-analysis-structure` is to analyze the phrase and to construct a structure grouping subphrases candidates to share the same chord. This grouping structure is built using background musical knowledge. A structure is composed by a sequence of `nodes`. For each `node` a collection of notes forming a subphrase and a main note is selected.

After the construction of the analysis structure, the harmonization process starts properly. The `propose-harmony` task is performed by a case-based method that proposes a chord for each node of the analysis structure. Moreover, when the case-based method is not able to find precedents for a given node, GYMEL uses a method for proposing a chord based on background musical knowledge.

The case-based method is decomposed in two subtasks: `retrieve` and `reuse`.

The `retrieve` task is performed, in turn, by a method decomposed following the usual retrieval subtasks: `identify`, `search`, and `select` subtasks.

The `identify` task is performed by a method, called `build-node-perspective`, based on perspectives (see Section 4.3). The `build-node-perspective` method builds a retrieval pattern based on the analysis structure of the phrase. Given a node, the method builds a perspective of that node taking into account the main note of the node and the selected chords of its two previous nodes. Specifically, the `build-node-perspective` method is defined as follows:

```
(define (perspective build-node-perspective)
  (node )
  (pattern (define (node)
             (main-note (define (note)
                          (pitch (define (relative-pitch)))))
             (prev (define (node)
                     (chord (define (chord)
                              (kind (define (chord-kind)))))
                     (prev (define (node)
                             (chord (define (chord)
                                      (kind (define (chord-kind))))))))))))
```

For instance, the application of that method to a node with a main note with pitch P4 and with two previous nodes with selected chords IImin7 and Imaj, yields the following pattern:

```
(define (node)
  (main-note (define (note)
               (pitch P4)))
  (prev (define (node)
          (chord (define (chord)
                   (kind IImin7)))
          (prev (define (node)
                  (chord (define (chord)
                           (kind Imaj))))))))
```

The `search` task is performed by a method based on the `retrieve-by--pattern` built-in method. This method retrieves, using the pattern constructed by perspectives, nodes sharing the same main note and the same chord progression of the two previous nodes than the node problem.

The `select` task performs a random selection. The Noos backtracking mechanism assures that all precedents will be attempted.

The goal of the `reuse` task is to access to the chord of the selected precedent node and propose a new chord, with same `kind` of the precedent chord, for the current node. The other features are determined from the features of the current node and background musical knowledge.

The GYMEL's problem solving method is exhaustive: all alternative solutions that it can be built are shown to the user. Then, the user chooses some of them to be stored into the episodic memory, and thus to be used in solving future problems.

6.5 Saxex

Saxex [Arcos, 1996] [Arcos et al., 1997b] is a case-based reasoning system for generating expressive performances of melodies based on examples of human performances that are represented as structured cases. *Saxex* has been developed using sound analysis and synthesis techniques of the SMS environment (Spectral Modeling Synthesis) [Serra, 1997] and the Noos language.

An input for *Saxex* is a musical phrase described by means of a musical score (a MIDI file) and a sound file. The score contains the melodic and the harmonic information of the musical phrase. The sound file contains the recording of an inexpressive interpretation of the musical phrase played by a musician. The output of the system is a new sound file, obtained by transformations of the original sound file, containing an expressive performance of the same phrase.

Solving a problem in *Saxex* involves three phases: the *analysis* phase, the *reasoning* phase, and the *synthesis* phase (see Figure 6.5).

Analysis and synthesis phases are implemented using SMS sound analysis and synthesis techniques, based on spectrum models, that are useful for the extraction of high level parameters from real sounds, their transformation and the synthesis of a modified version of the original sound. *Saxex* uses SMS in order

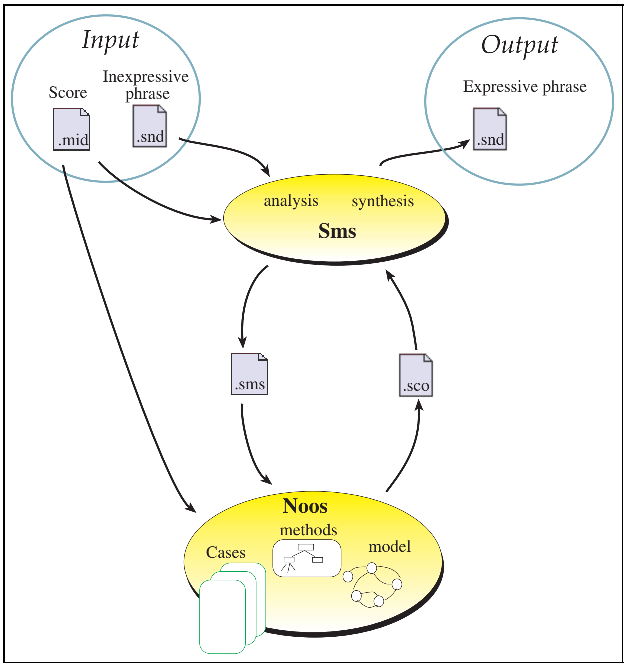


Figure 6.5. General view of Saxex components. Analysis and synthesis phases are performed in SMS. Reasoning phase is performed in Noos.

to extract basic information related to several expressiveness parameters such as dynamics, rubato, vibrato, and articulation. The SMS synthesis procedure allows *Saxex* the generation of new expressive interpretations (new sound files).

The reasoning phase is performed using case-based techniques and is implemented in Noos. This phase of *Saxex* incorporates background musical knowledge based on Narmour's Implication/Realization model [Narmour, 1990] and Lerdahl and Jackendoff's Generative Theory of Tonal Music (GTTM) [Lerdahl and Jackendoff, 1993]. These theories of musical perception and musical understanding are the basis of the computational model of musical knowledge of the system: using Noos perspectives methods with background musical knowledge, *Saxex* takes dynamical decisions about the relevant aspects of a given problem. That is to say, the background musical knowledge is used by *Saxex* as a set of declarative biases for retrieval.

Problems to be solved by *Saxex* are represented as complex structured cases embodying knowledge about the score of the phrase, knowledge about musical understanding of the phrase, and knowledge about the expressive performance of the phrase.

Figure 6.6. A Noos browser of the score for the ‘All of me’ ballad. Features are represented as thin boxes, dots indicate not expanded terms, and gray boxes express references to existing terms.

6.5.1 Modeling musical knowledge

The domain ontology of *Saxex* is composed of concepts representing three different types of musical knowledge: (1) concepts related to the score of the phrase such as notes and chords, (2) concepts related to background musical theories such as implication/realization structures and GTTM's time-span reduction nodes, and (3) concepts related to the performance of musical phrases. A *Saxex* case is represented as a feature term with three features: the `score`, the `structure`, and the `performance`.

The score

A score (see Figure 6.6) is represented embodying a musical `phrase`. A `phrase` is a feature term with two features: the `melody` feature, embodying a sequence of notes, and the `harmony` feature, embodying a sequence of chords (see Figure 6.6). Each `note` holds in turn a set of features such as the `pitch` of the note (`C5`, `G4`, etc), its `position` with respect to the beginning of the phrase, its `duration` (using CommonMusic notation [Taube, 1991][Taube, 1996]), a reference to its `underlying-harmony`, and a reference to the `next` note of the phrase. Moreover, a note holds the `metrical-strength` feature, inferred using GTTM theory, expressing the note's relative metrical importance into the phrase. Chords have also a set of features such as the `name` of the chord (`Cmaj7`, `E7`, etc), their `position`, their `duration`, and a reference to the `next` chord.

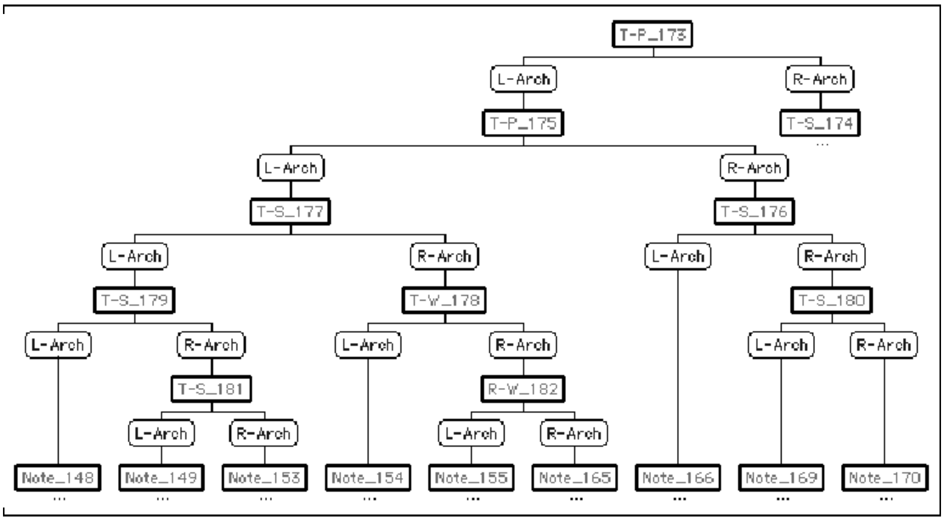


Figure 6.7. A Noos browser of the prolongational reduction structure for the 'All of me' ballad.

The musical structure

The musical structure embodies the musical analysis of the phrase built using the background musical knowledge. Narmour's implication/realization model (IR) proposes a theory of cognition of melodies based on eight basic structures. These structures characterize patterns of melodic implications that constitute the basic units of the listener's perception. Other parameters such as metric, duration, and rhythmic patterns emphasize or inhibit the perception of these melodic implications. The use of the IR model provides a musical analysis based on the structure of the melodic surface.

On the other hand, Lerdahl and Jackendoff's generative theory of tonal music (GTTM) offers an alternative approach to understanding melodies based on a hierarchical structure of musical cognition. GTTM proposes four types of hierarchical structures associated with a piece. This structural approach provides the system with a complementary view for determining relevant aspects of melodies.

The musical analysis builds a set of structures over the musical phrase. It is represented by the `analysis` feature term with three features: `prolongational-reduction`, `time-span-reduction`, and `process-structure`. The `prolongational-reduction` feature embodies a hierarchical structure describing tension-relaxation relationships among groups of notes. Tension-relaxation relationships are represented in Noos as trees (see Figure 6.7). The `time-span-reduction` feature embodies another hierarchical structure that describes the relative structural importance of notes within the heard rhythmic units of a phrase. These structural relationships are also represented in Noos

Figure 6.8. A Noos browser of the musical performance structure for the 'All of me' ballad.

as trees. Finally, feature `process-structure` embodies a sequence of implication/reduction (IR) Narmour's structures. There are eight types of IR structures. Each IR structure has a set of features representing the different roles that can play the notes in the structure (such as `first-note`, `med-notes`, and `last-note`) and characteristics specific of each IR structure such as the melodic direction.

The musical performance

A musical performance is represented as a sequence of events (see Figure 6.8). There is an `event` for each note within the phrase embodying knowledge about expressive parameters applied to that note. Specifically, an `event` has features representing expressive parameters of notes such as `dynamics`, `rubato`, `vibrato` level, `articulation`, and `attack`. Expressive parameters are described using qualitative labels as follows:

Changes on dynamics are described relative to the average loudness of the phrase by means of a set of five ordered labels. The middle label represents average loudness and lower and upper labels represent respectively, increasing or decreasing degrees of loudness.

Changes on rubato are described relative to the average tempo, also by means of a set of five ordered labels. Analogously to dynamics, qualitative labels about

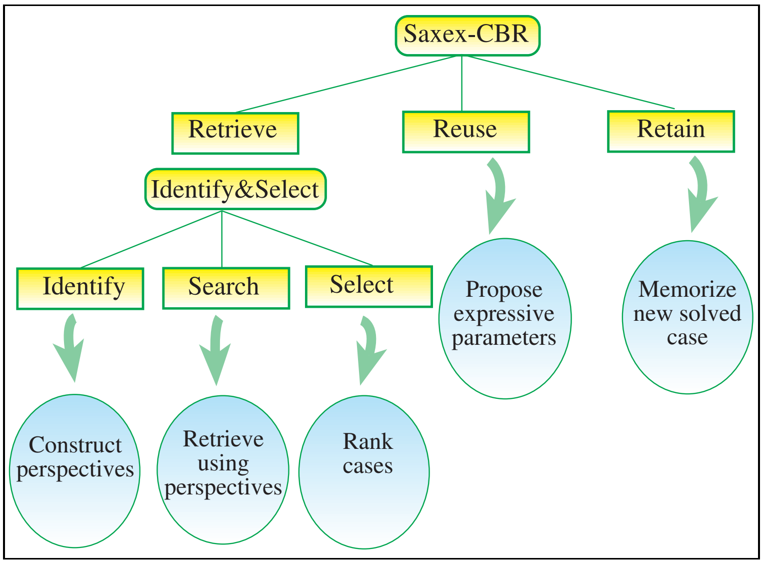


Figure 6.9. Task decomposition of the *Saxex* CBR method.

rubato cover the range from a strong accelerando to a strong ritardando.

The vibrato level is described using two parameters: the frequency vibrato level and the amplitude vibrato level. Both parameters are described using five qualitative labels from `no-vibrato` to `highest-vibrato`.

The articulation between notes is described using again a set of five ordered labels covering the range from `legato` to `staccato`.

Finally, *Saxex* distinguishes two transformations over a note attack: (1) reaching the pitch of a note starting from a lower pitch, and (2) increasing the noise component of the sound. These two transformations were chosen because they are characteristic of saxophone playing but other transformations can be introduced without altering the system.

6.5.2 The Saxex task

Given a musical phrase, *Saxex* infers a specific set of expressive transformations to be applied to every note in the phrase. These sets of transformations are inferred note by note. For each note in the phrase the same problem solving method is performed.

The problem solving method developed in *Saxex* for this purpose follows the usual subtask decomposition of CBR methods described in Section 4.5: *retrieve, reuse, and retain* (see Figure 6.9). Given a current note problem of a problem phrase, the overall picture of the subtask decomposition of *Saxex* method is the following:

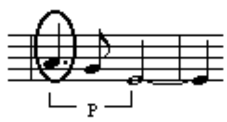
- *Retrieve*: The goal of the retrieve task is to choose the set of notes (cases) most similar to the current note problem. This task is decomposed in three subtasks:

- *Identify*: The goal of this task is to build retrieval perspectives using two alternative biases. A first bias based on Narmour's implication/realization model, and a second bias based on Lerdahl and Jackendoff's generative theory.
- *Search*: The goal of this second task is to search cases in the case memory using Noos retrieval methods and previously constructed perspectives.
- *Select*: The goal of the select task is to rank the retrieved cases using Noos preference methods. The preference methods use criteria such as similarity in duration of notes, harmonic stability, or melodic directions.

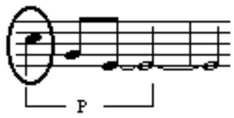
- *Reuse*: the goal of the reuse task is to choose a set of expressive transformations to be applied to the current problem from the set of more similar cases. The first criterion used is to adapt the transformations of the most similar case. When several cases are considered equally similar, transformations are selected according to the majority rule. Finally, when previous criteria are not sufficient, all the cases are considered equally possible alternatives and one of them is selected randomly.

- *Retain*: the incorporation of the new solved problem to the memory of cases is performed automatically in Noos. All solved problems will be available for the reasoning process in future problems.

After describing the task decomposition of *Saxex* problem solving method, we will introduce a simplified example to help its understanding. Let us suppose that *Saxex* has to infer a set of expressive transformations for the following note within a phrase[5]:

```
(define (note Note1)
  (pitch A4)
  (position 17)
  (duration Q.)
  (metrical-strength extremely-high)
  (belongs-to (define (P)
                (first-note (>>))
                (med-notes Note2)
                (last-note Note3)
                (direction down)))
  (next Note2))
```

The first task engaged is the `retrieve` task. The `retrieve` task engages in turn the `identify` subtask. Taking as example the following bias based on Narmour's model:

[5]The right side presents a picture of the note in musical notation.

```
(define (note P1)
  (pitch C5)
  (position 0)
  (duration Q)
  (metrical-strength extremely-high)
  (belongs-to (define (Process)
                (first-note (>>))
                (med-notes Note102)
                (last-note Note103)
                (direction down)))
  (next Note102))
```


```
(define (note P2)
  (pitch E4)
  (position 25)
  (duration Q)
  (metrical-strength extremely-high)
  (belongs-to (define (Process)
                (first-note (>>))
                (med-notes Note21)
                (last-note Note22)
                (direction up)))
  (next Note21))
```

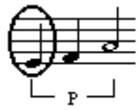



Figure 6.10. Two precedent cases retrieved by *Saxex* Problem solving method.

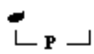
Determine as relevant the role that a given note plays in a implication/realization structure.

described as a Noos description as follows:

```
(define (note)
  (belongs-to (define (N-structure)
                ($f (>>)))))
```

We obtain the following constructed perspective of `Note1`:

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



that is, the first note of a `P` process.

Then, the *search* task is engaged in order to find similar situations among the precedent cases. Let us assume that the *search* task finds the following two notes (called `P1` and `P2`) as precedent cases (see Figure 6.10).

Next, the `select` task is engaged for ranking the precedents. Taking as preference criterion the melodic direction, precedent `P1` is considered as the most relevant precedent (since it belongs to a process with descending `direction` like `Note1`).

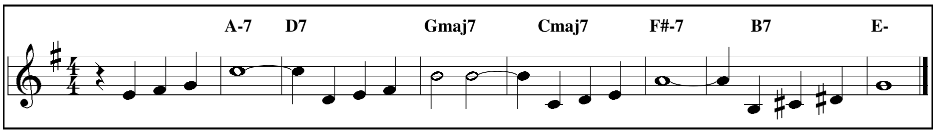


Figure 6.11. First phrase from the 'Autumn Leaves' theme.

After choosing precedent `P1` as the most relevant precedent, the `reuse` task is engaged. For this simplified example, since we only have selected one precedent, the set of expressive transformations to be applied to `Note1` are the same were applied to precedent `P1` and that are stored as part of precedent case `P1`.

6.5.3 Experiments

We have studied the issue of musical expression in the context of tenor saxophone interpretations. We have done several recordings of a tenor sax performer playing several Jazz standard ballads ("All of me", "Autumn leaves", "Misty", and "My one and only love") with different degrees of expressiveness, including an (almost) inexpressive interpretation of each piece. These recordings are analyzed, using the SMS spectral modeling techniques, in order to extract basic information determining the expressive parameters. The set of extracted parameters together with the scores of the pieces constitute the set of structured cases of the case-based system. From this set of cases and using similarity criteria based on background musical knowledge, the system infers a set of possible expressive transformations for a given piece. Finally, using the set of inferred transformations and the SMS synthesis procedure, *Saxex* generates a new sound file containing expressive performances of the jazz ballads.

We have performed two sets of experiments combining the different Jazz ballads recordered. The first set of experiments consisted in using examples of expressive performances of some phrases of a piece in order to generate new expressive performances of another phrase of the same piece. More concretely, we have worked with three different expressive performances of two phrases of a piece, having about twenty notes, in order to generate new expressive performances of another phrase of the same piece. This group of experiments has revealed that *Saxex* identifies clearly the relevant cases even though the new phrase introduces small variations with respect to the phrases existing in the memory of cases.

The second set of experiments consisted in using examples of expressive performances of some pieces in order to generate expressive performances of other pieces. More concretely, we have worked with three different expressive performances of pieces having about fifty notes in order to generate expressive performances of new twenty note phrases. This second group of experiments has revealed that the use of perspectives allows to identify situations such as long notes, ascending or descending melodic lines, etc. Such situations are also usually identified by a human performer.

Let as describe briefly some of the expressive transformations applied to the first phrase of the 'Autumn Leaves' theme (see the score in Figure 6.11) based on precedent cases of similar phrases. Concerning to changes of dynamics, the ascending melodic progressions are performed using crescendo. For instance, the first note of the theme (E) starts piano and the dynamics is successively increased yielding a forte in the fourth note (C). Concerning rubato, after the fourth note (C) the attack of the fifth note (D) is delayed and brought closer to the next note, then the duration of sixth note (E) is expanded, and finally the duration of the next note (F) is reduced. Vibrato is applied over notes with long duration combined with a dynamics decay (for instance, over fourth note). In ascending melodic progressions, articulation is also transformed by decreasing the interruption between notes (i.e. playing closer to legato than to staccato). Finally, the transformation of the attack consisted in reaching the eighth and ninth notes (B and B) starting from a lower pitch [6].

6.6 NoosWeb

The goal of the development of NoosWeb [Martín, 1996] was to provide a WWW user interface to Noos applications supporting the same interaction capabilities provided in the Noos window-based graphical user interface developed for MacOS computers (see Appendix A).

In order to provide full access to the Noos facilities, NoosWeb was designed to support, at least, (i) a Noos listener and (ii) a graph browsing facility. The Noos listener permits the evaluation of any valid Noos expression. The graph browsing facility emulates Noos browsers generating HTML documents.

The WWW interface to Noos is a sequence of dynamically-generated HTML documents that include forms (for engaging actions) and tables (for graph browsing). The user sees a standard NoosWeb interface document displaying a set of forms implementing valid actions (such as browsing and evaluating an expression from the listener) and displaying the answers of the last action performed. Although the user can view past actions using the NoosWeb client-cached documents, in order to avoid time-traveling problems, the user can only perform a new action from the current (the last) document. Actions requested from cached documents are detected and dismissed. Nevertheless, NoosWeb incorporates in documents the list of browsers displayed in the session allowing to redisplay any of them at any time.

NoosWeb is accessible at <http://www.iiia.csic.es/Interficies/NoosWeb>.

6.6.1 The NoosWeb architecture

The NoosWeb architecture is composed of two elements: a collection of clients connected through the network to a server (see Figure 6.13). The server side

[6] The reader can visit our web site at <http://www.iiia.csic.es/Projects/music/Saxex> for sound examples.

Figure 6.12. A NoosWeb browser of a sponge-problem from the SPIN system.

Figure 6.13. The NoosWeb architecture (from [Martín, 1996]).

maintains the state for each user (client) involved in a session. The server checks each client request against the current state. The server connects one or several Noos applications. Each Noos application only supports one user session over the WWW. A standard HTTP daemon uses CGI to process requests by means of a script called *NoosWebCGI*. *NoosWebCGI* dispatches requests to several Noos applications by TCP/IP. Each Noos application keeps contact with a Common-Lisp program called *NoosWebCL* that receives the requests, asks Noos when necessary, and returns an answer generating an adequate HTML document.

NoosWebCGI

NoosWebCGI is a CGI script implemented in the C language. *NoosWebCGI* is a dispatcher that distributes incoming requests to the appropriate Noos application by means of BSD sockets. *NoosWebCGI* is generic since it forwards all the request contents to the Noos application without any pre-processing and it also returns answers without any post-processing. *NoosWebCGI* maintains information about active Noos applications and current client sessions. When a user initiates a session, *NoosWebCGI* allocates a session number for an user and a Noos application. For each client request the script checks that the session identifier is valid. A request is valid if no time limit has been exceeded. There are time limits for session (1 hour) and for under-using (5 minutes without any request from a client). User sessions declared invalid are deallocated.

NoosWebCL

NoosWebCL is implemented in Common Lisp and is composed of four modules: *PassiveTCP*, *Parser-URL-encode*, *NoosWeb-obj*, and *Noos-to-HTML* modules. The module *PassiveTCP* waits to receive requests by means of TCP from *NoosWebCGI*. Received requests are parsed by the *Parser-URL-encode* module that determines which functionality of Noos is invoked (browsing a feature term, asking a specific query expression, evaluating a Noos description, etc). The *Parser-URL-encode* module engages Noos using the set of remote Noos functions. The result of the requested action is translated and stored into the *NoosWeb-obj* module structures. Finally, the *Noos-to-HTML* module generates an adequate HTML document that is returned to *NoosWebCGI*.

Changes to Noos

Noos has not been changed for allowing WWW interface. Only two new functions has been developed in order to provide an interface to remote calls. The first function is the *remote-browse* function that returns a Noos browsing structure in a list syntax. The second function is the *remote-eval* function that checks it is a legal Noos expression (a description of a query expression) and evaluates the expression returning the result of the evaluation.

6.7 Summary

This chapter presented a set of diverse applications developed using Noos by several persons at the IIIA. The purpose was to provide examples of how applications can be developed using Noos. In this chapter our description has focused on three aspects of the applications: how the domain knowledge required in each application is modeled using Noos representation capabilities, which problem solving methods are developed, and which learning methods are incorporated. Specifically we presented,

- how diverse domain specific case-based reasoning techniques are developed in Noos. Case-based reasoning is modeled by domain specific methods from a knowledge modeling analysis of an application. These methods incorporate domain-knowledge into Noos retrieval built-in methods. Examples of applications incorporating case-based reasoning methods are CHROMA, SPIN, GYMEL, and *Saxex*.

- how preferences are used for ranking precedents in case-based reasoning modeling different domain specific criteria in applications such as CHROMA and *Saxex*. The development and combination of diverse domain specific preferences provide a mechanism for the assessment of complex structured cases.

- how diverse inductive learning methods such as INDIE, DISC, and LID are developed in Noos. These methods are based on a search in the space of feature terms. Different methods perform several search strategies using the subsumption ordering in the feature terms space.

- how several alternative problem solving methods can be defined for solving the same task. We described how CHROMA uses four different methods for recommending purification plans in the *purification* task, and how SPIN uses three different methods for achieving the *identification* task.

- how CHROMA is provided by a metalevel reasoning method to decide, on a case-by-case basis, the applicability of the problem solving methods defined for solving the purification task and the order in which they are attempted.

- how case-based reasoning and inductive learning have been integrated in CHROMA and SPIN applications. The knowledge modeling analysis of a specific application determine how different learning methods can be integrated in different subtasks. Then, metalevel reasoning capabilities and the episodic memory of Noos allows to effectively implement this integration.

- how to use domain knowledge intensively. For instance, SHAM makes an intensive use of background musical knowledge for harmonizing melodies. SHAM models musical knowledge by means of (1) PSMs that identify which notes are important in the melody, (2) PSMs that propose alternative sets of feasible chords according to these notes, and (3) PSMs that combine the alternatives constructing a complete chord sequence.

- how the structured representation of cases offers the capability of treating subparts of cases as full-fledged cases. For instance, in GYMEL and *Saxex* applications the solution for a new problem is built by combining and adapting subparts of solutions from several precedent cases.

- how two complex musical theories for musical perception and musical understanding are modeled in *Saxex* and are then used for analyzing musical phrases and assessing their similitude with respect to other phrases.

- how perspectives are used as a mechanism to describe declarative biases for case retrieval in structured representations of cases. For instance, perspectives provide to GYMEL and *Saxex* applications a flexible way to dynamically construct partial descriptions for retrieval.

Finally, we presented NoosWeb, a WWW interface to Noos applications supporting the same interaction capabilities provided in the Noos window-based graphical interface developed for MacOs computers. A set of small improvements has been performed in Noos to support remote calls.

Chapter 7

Conclusions and future work

This thesis addressed the design and implementation of a representation language for developing knowledge systems that integrate problem solving and learning.

We have developed Noos, a reflective object-centered representation language for integrating inference and learning components in a uniform representation.

7.1 The Noos language and feature terms

Noos a representation language close to knowledge modeling frameworks, based on the task/method decomposition principle and the analysis of models required and models constructed by problem solving methods. This capability allows Noos to take advantage of the KA methodologies and libraries developed in KM frameworks.

The Noos modeling framework is based on four knowledge categories: domain knowledge, problem solving knowledge, episodic knowledge, and metalevel 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.
- Problem solving in Noos is considered as the construction of the *episodic model* of a problem. This model is obtained from transformations of problem data performed by problem solving knowledge. Episodic models built in solving problem tasks constitute the episodic knowledge of the system.
- Problem solving knowledge specifies a set of tasks and methods that construct a model of a problem (solve a problem). For a given subtask there may be multiple alternative methods that may be capable of solving that

subtask in different situations. A method can be decomposed into subtasks that may be achieved by other methods.

- Metalevel knowledge specifies knowledge about domain knowledge, problem solving knowledge, and episodic knowledge. These models are formed by metalevel concepts, metalevel relations, metalevel tasks, and metalevel methods. Moreover, metalevel knowledge includes preferences to model decision making about sets of alternatives present in domain knowledge and problem solving knowledge.

Noos is an object-centered representation language based on *feature terms*. Feature terms are related to the research based on λN calculus [Dami, 1994] and *ψ-terms* [Aït-Kaci and Podelski, 1993] that propose formalisms to model relational and object-oriented programming constructs. Adapting the theoretical results of these formalisms to our purpose, feature terms provide a natural way to describe partial knowledge amenable to extension. Feature terms are interpreted as partial descriptions denoting sets of individuals in a given domain. This semantical interpretation of feature terms brings about an ordering relation among them. We call this ordering relation *subsumption*. The intuitive meaning of subsumption is that of informational ordering. We say that a feature term ψ subsumes another feature term ψ' when all information in ψ is also contained in ψ'—or in other words, ψ is more general than ψ'.

Noos incorporates a declarative mechanism for decision making about sets of alternatives called *preferences*. For instance, preferences are used as 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 also 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 methods in applications such as CHROMA and *Saxex*.

Inference starts in Noos when the user poses a query to the system by means of a query expression. A query expression engages a task `F(D)` to be solved by its corresponding method. 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.

Methods in Noos can be view as functions with named parameters and backtracking. We formally described the global inference process in our system using Descriptive Dynamic Logic [Sierra et al., 1996], a propositional dynamic logic that provides a general framework for describing and comparing reflective knowledge systems.

7.2 Memory and learning

Our goal has been to provide a representation language for developing knowledge systems with learning capabilities. This goal required that machine learning techniques had to be modeled inside our language. Our proposal is that learning methods are methods (in the sense of knowledge modeling PSM) that can be analyzed also by means of a task/method decomposition and a set of models required models constructed by learning methods.

Both problem solving methods and learning methods perform inference (viewed in Noos as constructing episodic models). Learning methods differ from PSMs in that they use episodic models (i.e. past solved problems). Different kinds of learning methods use episodic models in different ways (see below the different approach of induction versus lazy learning approaches). These solved problems can be provided by a teacher or can be problems previously solved by the system itself. For learning methods to be able to reason from problems solved by the system, part of the behavior of the system has to be reified and stored in the system. In Noos, the episodic memory stores this representation of part of the behavior of the system. Moreover, we have incorporated a collection of reflective operations for accessing to and inspecting the episodic memory contents.

Episodic memory: Problems solved in Noos are automatically memorized (stored and indexed) in the episodic memory and are amenable to be accessed and reused in solving new problems. The problem solving behavior is represented in Noos in terms of tasks, methods, metalevels, and preferences. Episodic memory is organized in episodic models. An episodic model is the *explanation* of the inference process engaged by Noos in solving 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 task. Introspective capabilities form the basis that allows Noos programs to reason about the system behavior.

Integrated problem solving and learning: Our approach to integrate problem solving and learning is based on the following: whenever some knowledge is required by a problem solving method, and that knowledge is not directly available, there is an opportunity for learning. We call those opportunities *impasses* and the integration of learning is realized by learning methods that are capable of solving these impasses.

We have modeled the integration of different symbolic learning techniques as methods that can be decomposed in three main common subtasks: *Introspection*, *Construction*, and *Revision*. This common 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. Specifically:

- The introspection 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" (precedents), i.e. they are interesting to learn from.

- The construction task uses the relevant past experience (resulting from introspection) to generate some new model or body of knowledge.
  Eager learning methods construct a new model to be used for a specific problem solving method (that will be applied to future problems). Examples of eager learning methods are induction and analytical learning methods.
  Lazy learning methods follow a problem-centered approach—i.e. directly building the episodic model of a current problem from episodic model(s) of (some) retrieved precedent(s). Examples of lazy learning methods are CBR methods.

- The revision task decides whether and how the system's knowledge is modified by the newly constructed model. In simple situations the new model replaces the old model. In more complex situations, the task has to estimate whether the new model does improve the overall performance of the system.

Feature terms provide a structured representation of knowledge. A problem to be solved by the system is represented as a collection of concepts with many relationships among them. This structured representation of precedents offer the capability of treating subparts of them as full-fledged cases. That is to say, a new problem can be solved using subparts of multiple precedents retrieved from the episodic memory. This requires that new introspective mechanisms have to be provided. We have developed three introspective mechanisms to access 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 mechanism for content-based access to the episodic memory; and *perspectives*, a mechanism to describe declarative biases for case retrieval in the structured representation of cases.

Content-based retrieval: 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. Several domain-specific retrieval methods have been developed in Noos applications for solving the introspection task.

Perspectives: 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. Perspectives provide Noos with a mechanism for dynamically constructing retrieval patterns that specify the relevant aspects of a given problem. For instance, perspectives provide to GYMEL and *Saxex* applications a flexible way to dynamically construct partial descriptions for retrieval.

7.3 Methods and applications

Using the representation capabilities of Noos for modeling domain knowledge, problem solving knowledge, and learning, several PSMs and learning methods have been developed and integrated in several applications.

Based on the task/method decomposition principle, problem solving methods that use domain knowledge intensively can be designed and implemented. For instance, in the SHAM application several PSMs have been developed making an intensive use of background musical knowledge for harmonizing melodies. SHAM models musical knowledge by means of (1) PSMs that identify which notes are important in the melody, (2) PSMs that propose alternative sets of feasible chords according to these notes, and (3) PSMs that combine the alternatives constructing a complete chord sequence.

Furthermore, two complex musical theories for musical perception and musical understanding are modeled in *Saxex* and are then used for analyzing musical phrases and assessing their similitude with respect to other phrases in episodic memory.

Noos provides a collection of basic mechanisms allowing the development of different symbolic learning methods such as inductive learning, CBR, and analytical learning:

- *Inductive learning methods* in Noos are 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. Several strategies have been developed for constructing inductive learning methods that follow different searching biases.

- *Case-based reasoning methods* in Noos are 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* in Noos are methods that given (1) a training example whose problem task has been solved by a problem solving method M and (2) an operationality criterion, they construct a new problem solving method M_{op} for solving that task and obeying the operationality criterion. Analytical learning methods are based on the Noos introspective capabilities for inspecting the methods used in subtasks of M for solving the training example.

Several persons at the IIIA have developed learning methods and integrated them to applications. For instance, several domain specific case-based reasoning methods have been developed in Noos and integrated to applications such as CHROMA, SPIN, GYMEL, and *Saxex*. Moreover, several inductive learning methods such as INDIE, DISC, and a lazy learning method (called LID) have been developed by Eva Armengol in Noos and also integrated to CHROMA and SPIN applications. Finally, an analytical method called PLEC has been also developed and presented in this thesis.

We also presented how different learning methods can be designed and integrated in a problem solving system. Specifically, the research work realized in [Armengol, 1997] provides examples of knowledge systems developed in Noos that integrate different learning methods such as case-based reasoning and inductive learning methods.

Noos has been implemented using Common Lisp and currently is running on several platforms—including a window-based graphical interface for the Macintosh version of Noos.

Finally, we want to remark that Noos has been used, and is also currently used, by several persons at the IIIA and by other people with collaboration with the IIIA to develop different applications that integrate several problem solving methods and several learning methods. Specifically, we have described five applications developed using Noos and a WWW interface to Noos applications supporting the same interaction capabilities provided in the Noos window-based graphical interface developed for MacOS computers:

- CHROMA, a system for recommending a plan for the purification of proteins from tissues and cultures using chromatographic techniques (developed by Eva Armengol);

- SPIN, a system for identifying specimens of *Geodiidae* marine sponge family (developed by Eva Armengol);
- SHAM, a system for assisting a non expert musician to harmonize melodies using background musical knowledge (developed by Martí Cabré);
- GYMEL, another system for assisting the harmonization of melodies that uses a case-based reasoning approach (developed by Jordi Sabater);
- *Saxex*, a case-based reasoning system for generating expressive performances of melodies based on examples of human performances that I have developed in the context of my M.Sc.Thesis in Computer Music (this application has been awarded with the "Swets & Zeitlinger Distinguished Paper Award" at the 1997 International Computer Music Conference); and
- NoosWeb, a WWW interface to Noos applications (developed by Francisco Martín).

7.4 Future work

From the work realized in this thesis several research lines appear to be sufficiently interesting to pursue. Some of them have already started with a preliminary results.

- A first research line is to extend the basic retrieval and subsumption mechanisms provided in Noos enriching the comparison on numbers and strings. The work of G. Kamp [Kamp, 1997] on the admissibility of concrete domains can be adapted to Noos for this purpose.
- Another extension of Noos is to provide an agent-based environment for the cooperation between different Noos applications and on an heterogeneous environment. We have already started the study of how different case-based reasoning agents can cooperate in solving problems. We are developing two modes of cooperation among CBR agents: *Distributed Case-based Reasoning* (DistCBR) and *Collective Case-based Reasoning* (ColCBR). Intuitively, in DistCBR cooperation mode an agent A_i *delegates its authority* to another *peer* agent A_j to solve a problem—for instance when A_i is unable to solve it adequately. In contrast, ColCBR cooperation mode *maintains the authority* of the originating agent: an agent A_i can transmit a mobile method to another agent A_j to be executed there. That is to say, A_i *uses the experience* accumulated by other peer agents while maintaining the control on *how* the problem is solved. These preliminary results has been presented on [Plaza et al., 1997] [Plaza et al., 1996a].
- A third research line is to explore and extend the Noos capabilities for learning from failure. The episodic model built in solving a problem task stores the collection of preference orders between alternative methods for

solving a specific subtask and the method that has been effectively used in that subtask. Using this information we can infer the set of methods that has been failed in achieving the task. Nevertheless, Noos does not maintain the inference processes involved in the evaluation of these failed methods. This kind of knowledge could be useful for detecting situations where a PSM is not able to find a solution and then, to avoid failures in future problems.

- Another research line already started is focused on the study of the degradation of the system performance when the cost of searching for related knowledge outweighs the benefit of applying this knowledge. This problem is called the *utility problem* in [Tambe et al., 1990] and [Minton, 1990]. Different strategies have been proposed for solving this problem and could be useful also to Noos applications. Specifically, the research on deletion strategies in the context of case-based reasoning such as in [Smyth and Keane, 1995] is a promising direction to follow.

- Finally, a fifth research line already started is focused on developing new learning methods. Specifically, we are exploring inductive methods for acquiring methods from examples (as in inductive logic programming).

Appendix A

The Noos development environment

This Appendix describes the Noos development environment. The Noos language is implemented using Common Lisp [Steele, 1990] and currently is running in several platforms. The main development platform is the MacOS (using MCL [Digitool, 1996]), but it is also available for Unix machines and PCs (both using Clisp[1]).

The purpose of this Appendix is to show some of the tools implemented in the Noos development environment for assisting the modeling and constructing of applications in Noos. Our purpose is not to provide a user manual for developing applications in Noos.

A.1 Defining feature terms in Noos

There are two ways of adding new terms in Noos: by defining descriptions in a file and then loading the file into the Noos environment[2] or by directly typing descriptions on the Noos listener. For instance, if we type the `person`'s description on the listener (as shown in Figure A.1) the returned value is `<Person>`, which is the "print-name" of the `person` feature term.

Lazy evaluation

Lazy (on demand) evaluation means that no expression is evaluated unless it is needed for some computation. A description can thus refer to the name of another description even if it is defined later—on the listener or on the file being loaded or on another file. However, refinement appearing on the root of a description do require the constituent to be already defined. The reason is that

[1] Clisp is a public domain Lisp available at <ftp://ma2s2.mathematik.uni-karlsruhe.de/pub/lisp/clisp>.

[2] Files can be loaded by using the `load` Lisp function in the Listener.

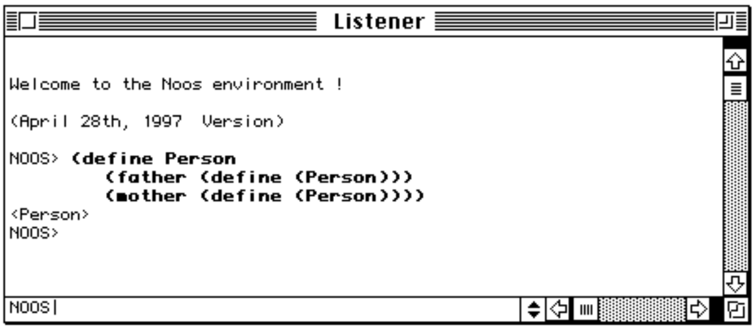


Figure A.1. Defining a new feature term on the Noos listener.

the outmost `define` in a description (the root) *is* evaluated when loading a file or typing on the listener. Anonymous descriptions appearing inside a description are not evaluated until needed. For this reason subdescriptions can refine a description that is defined later—or the same description being defined (see Figure A.1 where `father` and `mother` features of `person` are defined recursively by refinement of `person`).

Compact descriptions

In order to provide a more compact notation for the definition of closed methods in features, the syntax of Noos is extended. Using this extended syntax, a closed method defined by refinement of a built-in method can be defined using a position-based notation for specifying all of its required feature values.

The syntax used for specifying compact descriptions is the following:

(*built-in-name* param$_1$ $\cdots$ param$_n$)

where *built-in-name* is the name of a built-in method, and each param$_1$, $\ldots$, param$_n$ is a feature value, a path reference, or a compact description. Compact descriptions only can be used for specifying feature values.

The order of the required features is fixed by Noos. Appendix C describes the names of the built-in methods that accept compact descriptions and the order of specification of their required features.

For instance, the compact notation for the `identity?`, the `conditional`, and the `lower-than?` built-in methods is the following:

(`identity?` item1 item2)

(`if` condition result otherwise)

(`<` is-lower than)

where the names of the parameters indicate the specification order for the required features of each built-in method.

Using the compact notation, the following two closed methods defined in features `gas-gauge-reading` and `empty-level?`:

```
(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))))))
```

can be equivalently specified using the compact notation as follows:

```
(define Bills-car
  (owner Bill)
  (gas-level-in-tank 2)
  (gas-gauge-reading (if (< (>> gas-level-in-tank) 5)
                         empty
                         full))
  (empty-level?  (Identity?  empty (>> gas-gauge-reading))))
```

Note that the use of this compact notation requires that all required features have to be specified and that no other features can be specified. A second remark is that compact descriptions are specified using a single parenthesis in the same way as path references. Another remark is that using the compact notation we can only use compact descriptions for defining methods in the features—using the compact notation we can only define feature values using either constant values, path references, or compact descriptions. Moreover, since the name of the features are not specified, the position of the parameters determines the feature that a parameter is referred to.

The use of the compact notation allows also to define compositions of the same built-in method—such as arithmetic methods and methods for manipulating sets—in a easy way. For instance, since the `addition` built-in method has two required features, the sum of three numbers `a`, `b`, and `c`, using the compact syntax, has to be specified as the following composition:

```
(+ a (+ b c))
```

This compact syntax is extended for defining compositions as follows:

```
(+ a b c)
```

For instance, we can define the `earnings` of a specific company `comp` as the sum of the earnings of its three `production-lines` as follows:

```
(define (Company Comp)
  (line-1 (define (production-line)
            (line-name cosmetics)
            (earnings 12000)))
  (line-2 (define (production-line)
            (line-name toys)
            (earnings 23000)))
  (line-3 (define (production-line)
            (line-name nourishment)
            (earnings 8000)))
  (earnings (+ (>> earnings line-1)
               (>> earnings line-2)
               (>> earnings line-3))))
```

Multiple path references

Another language extension is to allow specifications of multiple path references in a feature. These multiple path references are interpreted using the union built-in method. In fact, the following description:

```
(define Person
  (uncles (>> brothers of mother)
          (>> brother of father)))
```

is a short syntax equivalent to the following description:

```
(define Person
  ((uncles (define (union)
             (item1 (>> brothers of mother))
             (item2 (>> brother of father)))))
```

Engaging inference

When a user types a path reference or an eval expression in the Listener, it is interpreted as a *query-expression* to be answered by the system. The Listener, however, does not accept relative path references. The reason is that a relative path reference only can be bound in the scope of a description. An alternative way to engage the inference is directly using the pop-up menus provided in the browsers (see Section A.4).

We have seen in Section 3.3.5 that, since reflective operations are references to feature terms, path references and reflective operations can be combined. These combined expressions are extended query-expressions that can be posed to the system in the listener. For instance, the following query-expression:

```
(meta (>> father of Tom))
```

asks for the metalevel of the person that is the `father` of `Tom`.

Whenever Noos yields a solution value for a problem task, the user can force backtracking to the Noos inference engine to search for another solution value for the task using the `force-backtracking` command.

For instance, let us assume that the following two descriptions have been added to Noos:

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

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

Then, we can pose the following query-expression to the listener:

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

The answer yielded by Noos will be `3344`. Next, forcing the backtracking with the `force-backtracking` command as follows:

```
(force-backtracking)
```

the inference is resumed in Noos yielding `8766`. If we force backtracking once more, the value yielded by Noos will be `Fail`—the token that Noos yields for indicating that no more values can be inferred.

A.2 The predefined sort hierarchy of Noos

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 (see Figure A.2):

- 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`,
- all built-in methods and the sort `method`, with the order relations `method` $\leq m$ for all built-in methods m,

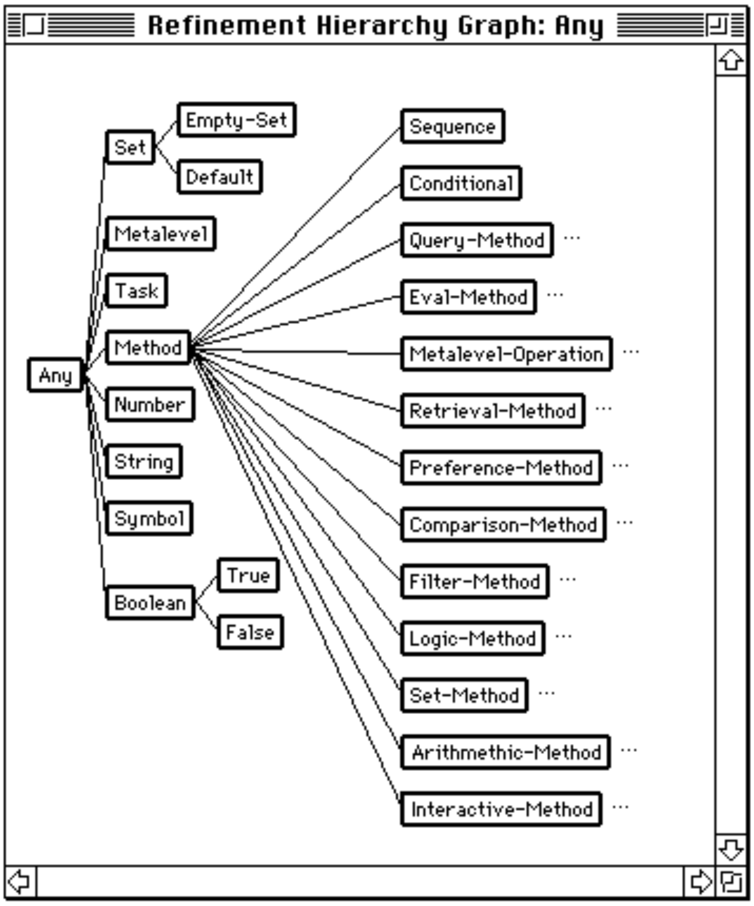


Figure A.2. The predefined sort hierarchy of Noos.

- sorts `metalevel`, `default`, and `task`.

The complete set of predefined built-in methods is described in Appendix D. From this set of initial sorts new sorts can be defined, using refinement, for specifying the sort hierarchy for a given domain.

A.3 Episodic memory

Whenever a problem task is solved by Noos, the user can decide to incorporate the episodic model built to the Noos episodic memory using the `freeze` command. For instance, after solving the `phone-number` of `Carol` we can type the following command:

```
(freeze Carol)
```

that will incorporate the episodic model built in solving `phone-number(Carol)` problem task into the episodic memory.

When an episodic model is incorporated to the episodic memory, no more inference can be engaged on this episodic model.

After solving a problem task, the user can decide not to incorporate the episodic model built in solving the task. For this situation, the Noos environment provides the `forget!` command. Specifying the `forget!` command over a feature term, the term is removed and will become inaccessible—and consequently no more inference can be engaged on this term.

Noos provides a way to explicitly determine that a specific feature term has not to be incorporated to the episodic model using the `:ephemeral` token as following:

`(define (`*constituent name* `) :ephemeral` body`)`

Specifically, the `freeze` command incorporates into the episodic memory all the feature terms belonging to an episodic model, excepting those explicitely specified as `:ephemeral`. For instance, the `:ephemeral` token is frequently used in retrieval patterns because a retrieval pattern is a term that is only constructed for retrieval purposes and it should not be retrieved in next problems.

Customizing Episodic memory

Since there are applications developed in Noos that will not require all the capabilities provided by the episodic memory of Noos, episodic memory can be customized for increasing the efficiency of the system.

A first option is to deactivate the episodic memory when the application does not use it. For instance, the SHAM application described in Section 6.3 solves problem tasks only using problem solving methods modeling background musical knowledge. Thus, episodic memory capabilities are deactivated in SHAM since they are not used.

Episodic memory can be deactivated using the `exclude` command with the `:all` token as follows:

```
(exclude :all)
```

Another option is to specify that some kind of terms are not to be incorporated into the episodic memory. For instance, there are applications that only require the retrieval of feature values and do not reason about the methods used in a specific task. These applications (such as *Saxex* CHROMA, SPIN, and GYMEL) do not require to introspect tasks, methods, and metalevels. The `exclude` command can also be used for this purpose, as follows:

```
(exclude task method metalevel)
```

The `exclude` command accepts any combination of some of the three different categories specified in the example: `task`, `method`, and `metalevel`. For instance, another legal option is the following:

```
(exclude task metalevel)
```

Finally, a third option is to specify that methods used in solving tasks can be removed after their evaluation. This option requires methods to be excluded to the episodic memory. The advantage of this option is that the Noos memory required by an application is lower. The disadvantage is that neither reflective operations nor browsers over the methods can be applied. The `compact-methods` command can be used for activate and deactivate this option using the following syntax respectively:

```
(compact-methods t)
(compact-methods nil)
```

A.4 Browsing

The Noos development environment provides four types of graphical browsing facilities: a *feature term browser*, a *task/method decomposition browser*, a *task structure browser*, and a *refinement hierarchy browser*.

There are two possibilities for browsing: using a text-based browser that can display into the listener or into a file, or using a window-based graphical interface. The first option is available for all the implementations of Noos. The window-based browsers are only available on the MCL implementation.

Text-based browsers and window-based browsers can be displayed typing specific browsing commands on the listener. Window-based browsers can be also displayed using the Noos menu developed for the MCL version.

A.4.1 Feature term browser

The feature term browser provides a graphical representation of a feature term. The `browse` command can be used to start a feature term browser using the following syntax:

(`browse` *name* [*depth*])

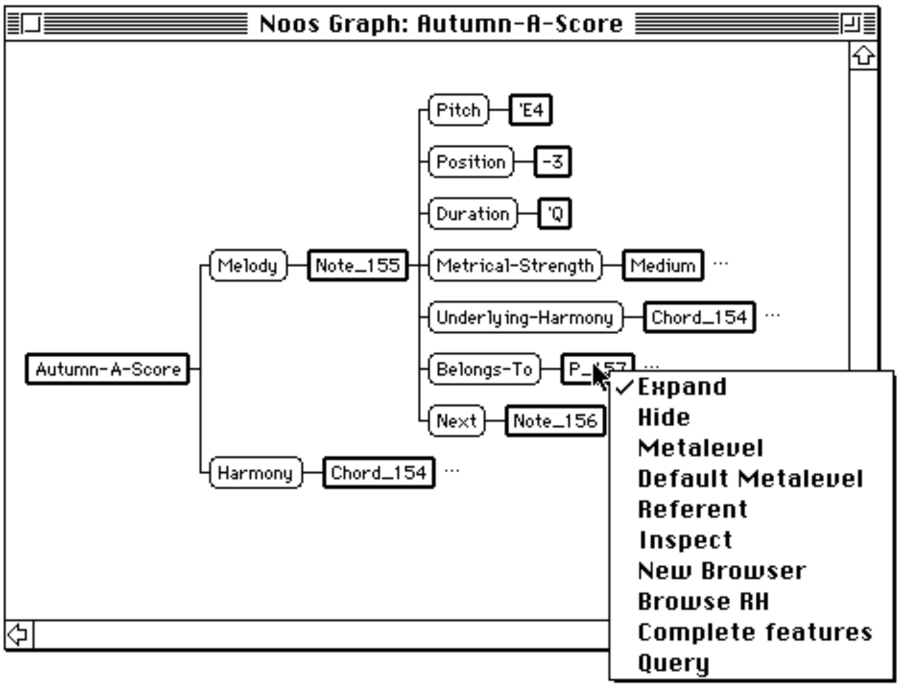


Figure A.3. A browser of the score of ‘Autumn Leaves’ ballad from *Saxex* application. A pop-up menu has been activated by clicking the `P_157` node.

where *name* is an identifier of a feature term and *depth* is an optional integer number indicating the depth level of the term to be displayed. The default value for the depth level for all the Noos browsers is 3.

For instance, a browser of the first phrase of the ‘Autumn Leaves’ ballad from *Saxex* application (described in Section 6.5) can be started by the following command:

```
(browse Autumn-A-Score)
```

obtaining the browser window of Figure A.3. Note that terms are displayed as graphs with nodes representing their identifiers as thick boxes; features are represented as thin boxes; ellipsis (three dots) indicate nodes amenable to be expanded; and gray boxes express references to existing nodes—nodes expanded in another place in the window.

Using the default value for the depth parameter (3), a feature term browser displays a root node (depth 1) and their features: the feature names (depth 2) and the feature values (depth 3). All feature values amenable to be expanded are indicated with ellipsis.

Many browsers can be open at the same time. Each browser is displayed in a different window. Browsers can be dynamically expanded using *pop-up menus*. Specifically, there are two kinds of pop-up menus: one for nodes and another for features (in Figure A.3 a pop-up menu for terms is shown).

The pop-up menu unfolded when the user clicks on a node box has the following ten commands:

- *Expand*: the *expand* command allows to expand a node's term displaying its features. The features are displayed without engaging the inference of their values (i.e. only computed and constant values are shown). When a feature value is neither a constant value nor it has been inferred, a node with label `Unknown` is displayed. The expand command performed on this unknown node will engage the inference of that feature value. The result yielded is a node with the value, if it can be inferred, or, otherwise, a node with the `FAIL` label.

- *Hide*: the *hide* command is the converse command to expand: the features of a node are hidden and ellipsis are displayed for indicating that the node is amenable to be expanded.

- *Metalevel*: the *metalevel* command starts a new browser displaying the metalevel term of the selected node. When the metalevel has not been defined, no new browser is started.

- *Default metalevel*: the *default metalevel* command starts a new browser displaying the default metalevel term of the selected node. When the default metalevel is not defined, no new browser is started.

- *Referent*: the *referent* command is the converse command to the *metalevel* command. The *referent* command starts a new browser displaying the referent term of the selected node. When the referent does not exist, no new browser is started.

- *Inspect*: the *inspect* command starts the inspector of Lisp. We have extended the Lisp inspector for displaying Noos terms.

- *New browser*: the *new browser* command starts a new browser displaying the selected node. This command is not allowed over the root node. If there is another browser where the selected node is the root, this browser window is activated and no new browser is started.

- *Browse RH*: the *browse RH* command starts a refinement hierarchy browser with the sort of the selected node as root (see Section A.4.4).

- *Complete features*: Following the lazy approach of inference in Noos, in definitions by refinement, features defined in the constituent are only incorporated to the refined term when they are needed for any computation. This means that when a term is expanded in a browser only features defined in the term and features inferred in some problem task will be displayed. The *complete features* command forces to display all the features defined. This option only displays features without engaging the inference for their values. Next, the user can use the *expand* command for engaging the inference in a specific feature.

- *Query*: finally, the query command on a node allows to engage the inference on any feature for that node by means of specifying a feature name.

The pop-up menu unfolded when the user clicks on a feature box has the following six options:

- *Expand*: the *expand* command, analogously to the previous pop-up menu, allows to expand a feature displaying its value. The value is displayed without engaging inference. When the feature value is neither a constant value nor it has been inferred, a node with label `Unknown` is displayed.
- *Hide*: the *hide* command is the converse command to expand: the feature value of a feature is hidden.
- *Task*: the *task* command starts a new browser displaying the task term of the selected feature.
- *Method*: the *method* command starts a new browser displaying the method term defined for the selected feature. When no method is defined, no new browser is started.
- *Task structure*: the *task structure* command starts a task structure browser displaying the episodic model constructed for inferring the value of the feature (see Section A.4.3).
- *Task decomposition*: the *task decomposition* command starts a browser displaying the task/method decomposition defined for the selected feature (see Section A.4.2).

The text-based version of a feature term browser can be obtained using the `tbrowse` command. For instance, the score shown in Figure A.3 yielded by the following command:

```
(tbrowse Autumn-A-Score 5)
```

obtaining the text-based browser presented in Figure A.4.

A.4.2 Task/method decomposition browser

The task/method decomposition browser provides a graphical representation of the recursive decomposition of a task into subtasks by means of methods. That is to say, this kind of browser provides a graphical representation for the problem solving knowledge (See Section 3.1). The `browse-task` command and the `browse-method` command can be used, respectively, to start a browser for a specific task or a browser for a specific method using the following syntax respectively:

```
{Autumn-A-Score
  (Melody {Note_155
             (Pitch E4)
             (Position -3)
             (Duration Q)
             (Metrical-Strength {Medium})
             (Underlying-Harmony {Chord_154})
             (Belongs-To P_157)
             (Next {Note_156})})
  (Harmony {Chord_154
             (Name Cmaj7)
             (Position 0)
             (Duration W)
             (Next {Chord_155})})}
```

Figure A.4. A text-based feature term browser of the score of 'Autumn Leaves' ballad from *Saxex* application.

(`browse-task` *task-name term-name* [*depth*])

(`browse-method` *method-name* [*depth*])

For instance, a browser of the task/method decomposition of `general-diagnosis` method following [Benjamins, 1993] can be started by the following command:

```
(browse-method general-diagnosis)
```

obtaining the browser window of Figure A.5. Tasks are drawn with thick boxes; methods are drawn with thin boxes; and ellipsis indicate the nodes amenable to be expanded.

Note that a method can be displayed using a feature term browser and using a task/method decomposition browser. Nevertheless, the information displayed in each browser is different. The first one shows the method's (sub)tasks and their values; the task/method decomposition browser shows the method's tasks and their (sub)methods.

Many task/method decomposition browsers can be open at the same time and each browsing is displayed in a different window. Browsers can also be dynamically expanded using *pop-up menus*. Similarly to feature term browsers, there are two kinds of pop-up menus: one pop-up menu for methods and another for tasks.

The set of commands allowed in methods are the same as before, excluding the *query* command: *expand*, *hide*, *metalevel*, *default metalevel*, *referent*, *inspect*, *new browser*, *browse RH*, and *complete features*.

The set of commands allowed in tasks are the following: *expand*, *hide*, *referent*, *inspect*, and *new browser*.

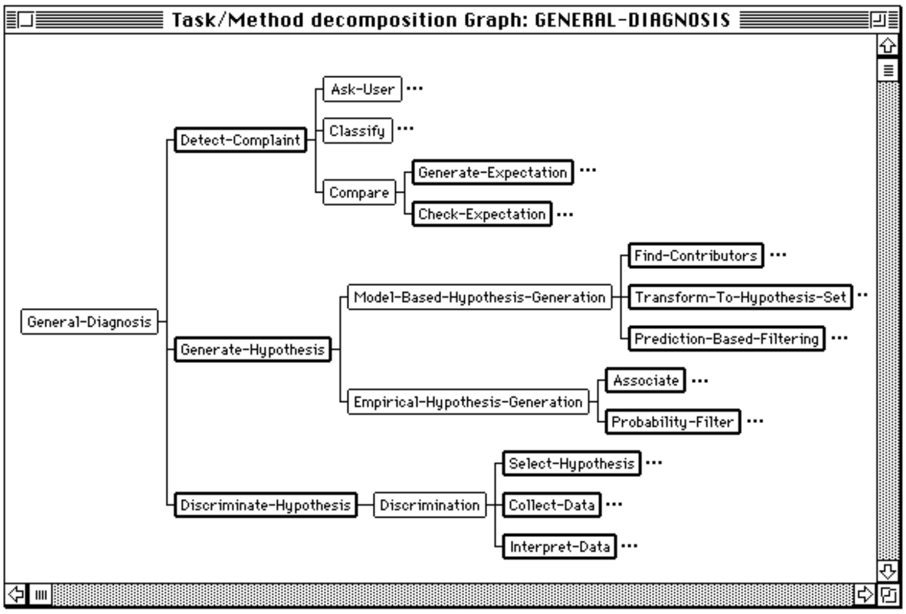


Figure A.5. A browser of the task/method decomposition for the general diagnosis method.

The text-based version of a task/method decomposition browser can be obtained using the `tbrowse-task` and the `tbrowse-method` commands. For instance, the task/method decomposition for the general diagnosis method shown in Figure A.5 yielded by the following command:

```
(tbrowse-method General-Diagnosis 3)
```

will be the following:

```
{General-Diagnosis
  (Detect-Complaint {Ask-User}
                    {Classify}
                    {Compare})
  (Generate-Hypothesis {Model-Based-Hypothesis-Generation}
                       {Empirical-Hypothesis-Generation})
  (Discriminate-Hypothesis {Discrimination})}
```

A.4.3 Task structure browser

The task structure browser provides a graphical representation of the part of the episodic model concerning the methods and subtasks effectively involved in

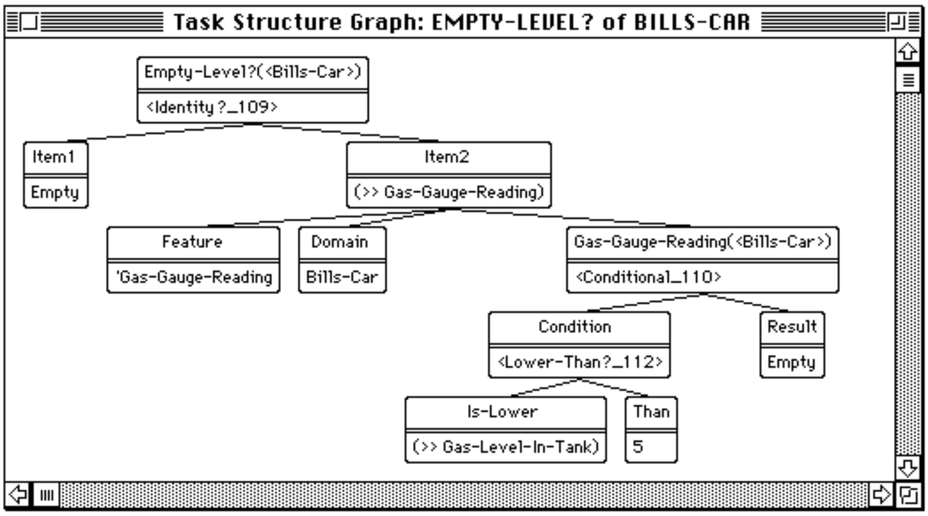


Figure A.6. A task structure browser from the episodic model of problem task `empty-level?(Bills-car)`.

solving a specific problem task. The `browse-stask` command can be used to start a browser for a specific task using the following syntax:

(`browse-stask` *feature-name term-name* [*depth*])

For instance, a task structure browser from the episodic model of problem task `empty-level?(Bills-car)` can be started by the following command:

```
(browse-stask empty-level?  Bills-car)
```

obtaining the browser window of Figure A.6. Each node in the browser shows a task and the method that has solved that task. The upper part shows the task name and the lower part shows the method printname.

Note that only the subtasks effectively involved in solving a specific method are displayed. For instance, the subtask decomposition of the `conditional_110` method of Figure A.6 is only composed of the `condition` task and the `result` task since the result of the `condition` task is `true`.

Another example of a task structure browser built after solving the `cup?(Obj1)` problem task can be found in Section 4.7.

Task structure browsers can also be dynamically expanded using *pop-up menus*. The pop-up menus have the following five commands: *expand*, *hide*, *referent*, *inspect*, and *new browser*.

Note that the referent of a task is the result value inferred by the evaluation of its method (see Section 3.3.5).

The text-based version of a task structure browser can be obtained using the `tbrowse-stask` command. For instance, the task structure from the episodic

```
{Saxex-sort}
   {Saxex-Problem}
   {Score-element}
      {Phrase}
      {Note}
      {Chord}
   {Analysis-Element}
      {Phrase-Analysis}
      {Nstructure}
      {Time-Span-Node}
      {Prol-Reduc-Node}
   {Performance-Element}
      {Phrase-Performance}
      {Event}
      {Label}
```

Figure A.7. A text-based feature refinement hirerarchy browser of the *Saxex* application.

model of problem task `empty-level?(Bills-car)` shown in Figure A.6 yielded by the following command:

```
(tbrowse-stask empty-level?  Bills-car 8)
```

will be the following:

```
(Empty-Level?(Bills-Car)
  {Identity?_109
    (Item1 {Empty})
    (Item2 {Infer-value (>> Gas-Gauge-Reading)
             (Feature 'Gas-Gauge-Reading)
             (Domain {Bills-Car})
             (Gas-Gauge-Reading(Bills-Car)
               {Conditional_110
                 (Condition {Lower-Than_112})
                 (Result {True})})})})
```

A.4.4 Refinement hierarchy browser

Finally, the refinement hierarchy browser provides a graphical representation of the sort hierarchy defined in developing a specific application refining the built-in sort hierarchy of Noos. The `browse-RH` command can be used to start a browser from a specific sort using the following syntax:

(`browse-RH` *sort-name* [*depth*])

For instance, the refinement hierarchy browser defined for the *Saxex* application can be started by the following command:

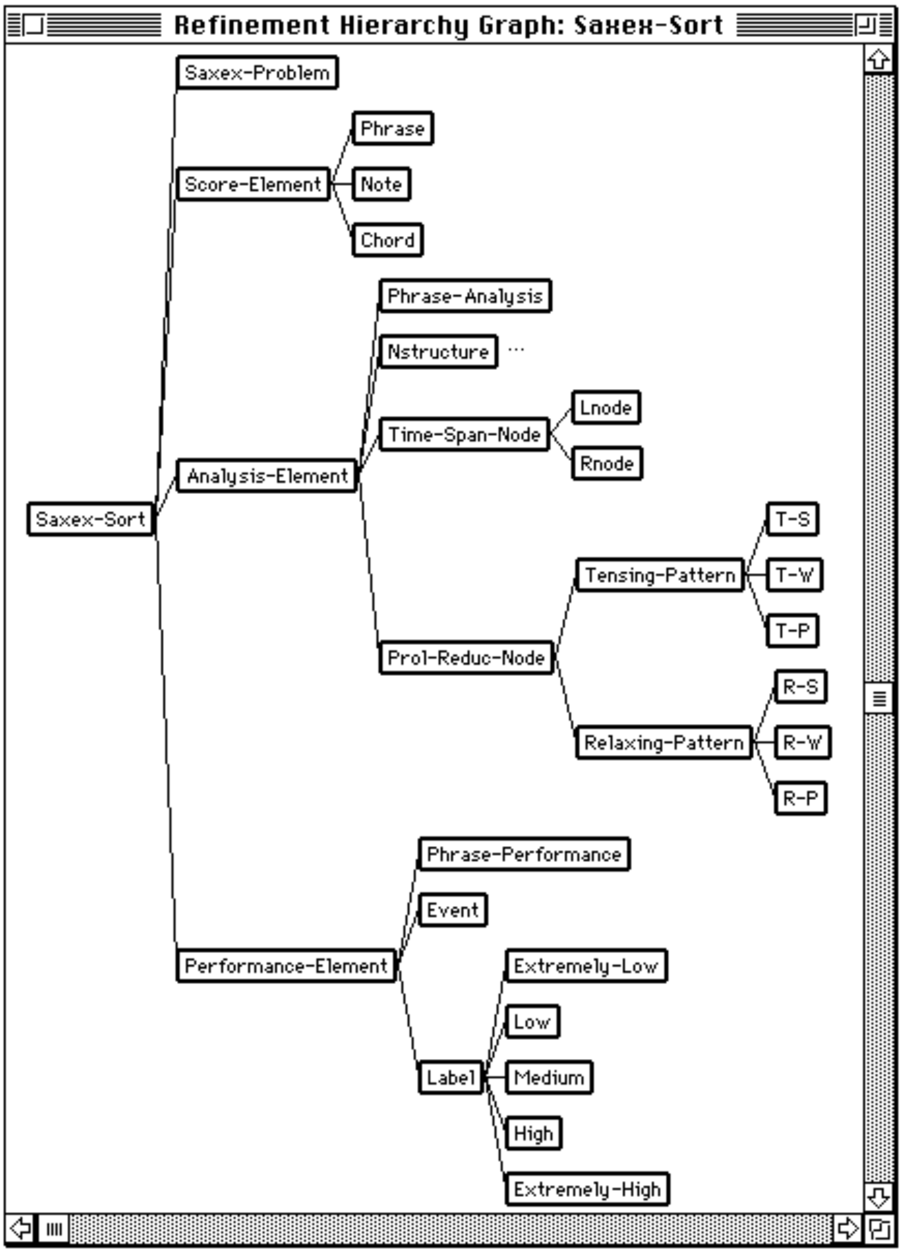


Figure A.8. A refinement hierarchy browser from the *Saxex* application.

```
(browse-RH Saxex-sort)
```

obtaining the browser window of Figure A.8.

This kind of browsers can also be dynamically expanded using *pop-up menus*. The pop-up menus have the following four commands: *expand*, *hide*, *inspect*, and *new browser*.

The text-based version of a refinement hierarchy browser can be obtained using the `tbrowse-rh` command. For instance, the refinement hierarchy browser defined for the *Saxex* application shown in Figure A.8 yielded by the following command:

```
(tbrowse-rh Saxex-sort 3)
```

obtaining the text-based browser presented in Figure A.7.

A.5 Tracing

The Noos development environment provides two commands for tracing the inference: `usual-trace` and `trace-feature`.

The `usual-trace` command enables a complete trace of the inference. The trace can be disabled using the `no-trace` command. Examples of Noos traces have already shown in Section 3.2.5 and in Section 3.5.

Figure A.9 shows the trace generated in solving the `empty-level?(bills-car)` problem task that has the episodic model displayed in Figure A.6.

The `trace-feature` command enables a selective trace on a set of feature names specified by the user. Since the complete trace generated in solving a complex problem task may be too large, the `trace-feature` trace option allows to trace only the representative features.

The `trace-feature` command is disabled using the `disable-trace` command. Both commands `usual-trace` and `trace-feature` can be used simultaneously.

For instance, the inference trace generated in solving problem task `empty-level?(bills-car)` with three features selected for trace as follows:

```
(trace-feature empty-level?  gas-gauge-reading gas-level-in-tank)
```

will be the following trace:

```
(>> empty-level?  of bills-car)

 1 Query: empty-level?(<bills-car>)
 2    Query: gas-gauge-reading(<bills-car>)
 3      Query: gas-level-in-tank(<bills-car>)
 4      Result: 2
 5    Result: <empty>
 6 Result: <true>

<true>
```

```
(>> empty-level? of bills-car)

Eval: <Infer-value (>> empty-level? of bills-car)>
⟹
   Task: empty-level?(<bills-car>)
   Eval: <identity?_109>
   ⟹
     Task: item1(<identity?_109>)
     Value: <true>
     Task: item2(<identity?_109>)
     Eval: <Infer-value (>> gas-gauge-reading)>
     ⟹
        Task: feature(<Infer-value (>> gas-gauge-reading)>)
        Value: 'gas-gauge-reading
        Task: domain(<Infer-value (>> gas-gauge-reading)>)
        Value: <bills-car>
        Task: gas-gauge-reading(<bills-car>)
        Eval: <conditional_110>
        ⟹
          Task: condition(<conditional_110>)
          Eval: <lower-than?_112>
          ⟹
             Task: is-lower(<lower-than?_112>)
             Eval: <Infer-value (>> gas-level-in-tank)>
             Value: 2
             Task: than(<lower-than?_112>)
             Value: 5
          ⟸
          Value: <true>
          Task: result(<conditional_110>)
          Value: <empty>
        ⟸
        Value: <Empty>
     ⟸
     Value: <Empty>
   ⟸
   Value: <true>
⟸
Value: <true>

<true>
```

Figure A.9. The trace generated in solving the `empty-level?(bills-car)` problem task.

A.6 Extending built-in methods

Noos provides a collection of basic built-in methods. Nevertheless, some applications may require specific elementary methods that are not provided. Noos allows a simple way to incorporate new built-in methods using the `Define-Built-In` macro and Lisp expressions. This macro allows to define a new built-in method by means of specifying a name for the method, the set of required features of the method, and a Lisp expression performing a specific combination of required features. The syntax is the following:

`(define-built-in` *name* `(`*param1* ... *paramn*`)` *lisp-expression*`)`

where *name* is the name of the new built-in method we are defining; *param1* ... *paramn* are the names of the required features for the built-in; and *lisp-expression* is the expression that implements the built-in method. Inside the lisp expression values of required features can be accessed by using the names of required features as variables.

For instance, the `subtract` built-in method is defined as follows:

```
(Define-Built-In Subtract (amount minus) (- amount minus))
```

Moreover, we have to specify the new description of the method in Noos defining the `subtract` term as follows:

```
(define (method Subtract)
  (amount )
  (minus ))
```

The definition of a new built-in method can be specified in a file and then loaded into the Noos environment.

Appendix B

Glossary

The purpose of this appendix is to provide a collection of definitions about the set of different concepts involved in Noos language as well as a reference to the section of the memory in which are defined.

In this glossary, terms appearing in **boldface** indicate they are defined in the glossary. Often we will use *term* as a shorthand for *feature term* (and will not appear in boldface).

Anonymous terms — A term that has no **identifier**, it can be referenced only by its position (as a subterm of other terms). For instance, in `(define (person Jack) (girlfriend (define (girl))))` the girlfriend of Jack is anonymous and can only be referenced by `(>> girlfriend of Jack)`. (§ 3.2.2).

Antiunification — A basic operation defined in Noos that given two terms constructs another term holding *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 terms yields a greatest lower bound with respect to **subsumption** ordering. (§ 4.6).

Any — The highest sort in the Noos refinement hierarchy. `Any` represents the minimum information and all the other sorts are more specific than `any`. See also **single description**. (§ 3.2.2).

Built-in method — A Noos predefined method. 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. (§ 3.2.4, § D).

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. (§ 3.1).

Constant term — A feature term that is not a **method**–i.e. is not evaluable. (§ 5.2).

Constituent — The constituent of a term `T` is the identifier of a term `T'` from which `T` has been constructed by **refinement**. (§ 3.2.2).

Current-task — A reflective operation that allows to refer to the task in which a method is involved. (§ 3.3.5).

Default — Reflective operator that allows to refer to the **default metalevel** term of a term. § 3.3.5.

Default metalevel — A special kind of **metalevel** that applies to all the features of its **referent**. In a default metalevel we can specify a method (or a set of methods) for *any* feature of a **referent**. (§ 3.3.2).

Description — The syntax Noos uses for constructing **feature terms**. The description syntax is based on lists (like Lisp) starting with token `define`. For instance, `(define (person man))`. (§ 3.2.1).

Episodic knowledge — The reification of part of the behavior of the system represented in Noos. Episodic knowledge is is organized in **episodic models** and stored in the **episodic memory**. (§ 3.1, § 4.1).

Episodic memory — Episodic memory is the (accessible and retrievable) collection of the episodic models of the problems that a system has solved. (§ 4.1).

Episodic model — The *explanation* of the inference process engaged by Noos in solving 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. (§ 4.1).

Ephemeral term — A term that is not memorized—and thus, it is not amenable to **retrieval**. (§ A).

Evaluable term — A kind of feature term that models Noos methods. Evaluable feature terms are interpreted as functions. (§ 5.9).

Failure — In a specific subtask, a failure occurs when no value for that task can be inferred—and causes backtracking at that point. When a global failure occurs, the token `fail` is returned. (§ 3.5).

Feature term — The basic data structure of Noos. They can be seen as a generalization of first order terms and lambda terms. They are extendable records organized in a **subsumption** hierarchy. Feature terms are constructed by means of **descriptions** and the **refinement** constructor. (§ 5.2, § 5.6).

Identifier — An identifier is a symbol denoting a term. Terms that have an identifier are **named terms**, otherwise they are **anonymous terms**. (§ 3.2.2).

Introspection — In Noos, the capability of accessing to and reasoning about the episodic memory.

Labeled graph representation — A term can be represented as a labeled directed graph that has, for each variable $X : s$, a node q labeled with sort s, and has an arc from q to another node q' labeled by f, for each feature f defined in q with feature value q'. (§ 5.7).

Matching — See **subsumption**.

Memorization — The property of **permanent terms** to be stored in **episodic memory** and be amenable to **retrieval**. (§ 4.1).

Meta — Reflective operator that allows to access to the metalevel term of a term. (§ 3.3.5).

Metalevel — A feature term in a metalevel relationship with a **referent** term `B`. The features of the referent B have a corresponding feature with the same name on the metalevel. A feature f of the metalevel has as feature value the set of methods methods that are *applicable* to the feature f of the referent B. (§ 3.3.1).

Metalevel knowledge — Knowledge about **domain knowledge**, **problem solving knowledge**, and **episodic knowledge**. Metalevel knowledge is formed by metalevel concepts, metalevel relations, metalevel tasks, and metalevel methods. Moreover, metalevel knowledge includes **preferences**. (§ 3.1, § 3.3).

Method — An evaluable feature term. Formally, a function that receives parameters by feature names. A method term is *closed* when it possess all required parameters. Methods are defined by refinement. (§ 3.2.4).

Named Term — A term with an **identifier**. A named term can be referenced by its identifier using `(>> of NamedTerm)` although syntactic sugar allows in some places to write only `NamedTerm`. (§ 3.2.2).

Node — See **labeled graph representation**.

Path reference — A list that starts with the `>>` token. There are two kinds of path references: absolute and relative path references. An example of an absolute path reference is `(>> symptom of car)`. An example of a relative path reference is `(>> price model)`. (§ 3.2.3).

Permanent terms — **Feature terms** that are memorized (see **memorization**) and, thus, amenable to **retrieval**. (§ A).

Perspective — A mechanism for describing declarative biases for Noos retrieval. (§ 4.3).

Preferences — Model decision making about sets of alternatives present in **domain knowledge** and **problem solving knowledge**. 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. (§ 3.4).

Problem solving knowledge — Problem solving knowledge specifies a set of tasks and methods that construct a model of a problem (solve a problem). For a given subtask there may be multiple alternative methods that may be capable of solving that subtask in different situations. A method can be decomposed into subtasks that may be achieved by other methods. (§ 3.1).

Problem task — A task engaged by a **query expression**. A problem task `F(D)` engages the inference to determine the feature value for feature `F` of feature term `D`. (§ 3.2.5).

Query expression — A question that is posed to the system. Usually, a query expression is asking for a feature value, as in `(>> diagnosis of patient-33)`. Since evaluation in Noos is lazy, no feature values are inferred until a query is performed—and only those feature values needed will be computed. (§ 3.2.5).

Reference — See **identifier** and **path reference**.

Referent — The referent of a **metalevel** is that term it is metalevel of. The referent of a **task** is the result value of that task. Furthermore, the referent reflective operator allows to access to the referent—if it exists—of a term. (§ 3.3.1, § 3.3.5).

Refinement — A constructor operation that builds a term from another (already defined) term. Refinement involves two distinct aspects: (1) code reuse (the construction of a term by reusing another term) and, (2) subtyping (the definition of a domain-specific **sort** hierarchy). (§ 3.2.2).

Reflective operator — An operation that allows to access and to inspect the **metalevel knowledge** (see operations namely **meta**, **default**, **task**, **current-task**, and **referent**). (§ 3.3.5).

Reification — The process by which a Noos expression is converted into an object (a feature term). Reification is performed by the **reify** construct. (§ 3.3.6).

Reify — A construct that takes a path reference or a compact description and builds an **method** that reifies it.

Retrieval — A mechanism for content-based access to the episodic memory. Noos provide a set of basic retrieval methods. Retrieval methods allow to retrieve previous relevant episodes from the **episodic memory** using relevance criteria. (§ 4.2).

Root — The root **node** of a feature term is the outmost **node** in a **description**. For instance, the root in the following description `(define (person :id Jack) (girlfriend (define (girl))))` is the node of sort `person` with identifier `Jack`. (§ 3.2.2).

Single description — Compact syntax for defining feature terms by refinement of the top sort **any**. A description such as `(define foo)` is just a short syntax equivalent to `(define (any foo))`. (§ 3.2.2).

Sort — A symbol that denotes a set of the individuals of a domain. Sorts form a collection of partially ordered symbols. A set of predefined sorts are defined in Noos. New sorts are defined using **refinement**. In Noos the top sort is called **any**. (§ 3.2.2, § 5.2.2, § A).

Subsumption — Informational ordering among feature terms. We say that a feature term ψ subsumes another feature term ψ' when all information in ψ is also contained in ψ'—or in other words, ψ is more general than ψ'. (§ 5.6).

Symbol — See **identifier**.

Task — Tasks reify the status of the inference in the language. A given task reifies the inference status for a feature of a term. 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). (§ 3.3.4).

Transparent methods — 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. This capability of Noos methods is used, for instance, in analytical learning methods. (§ 4.1).

Appendix C

The Noos Syntax

This Appendix describes the syntax of Noos using BNF notation. We write predefined terminal symbols, that are part of the Noos language, in `typewriter` font. We write user-defined identifiers in *italic* font. We write non-terminal symbols in normal type face. Symbols ::=, [,] |, *, + are part of the BNF formalism as follows:

L ::= R	defines the syntax of L as R
[X]	defines an optional item
X | Y	defines two alternative options X and Y
X+	defines one or more occurrences of X
X*	defines zero or more occurrences of X

top-level-expression	::=	description
	|	query-expression
description	::=	single-description
	|	named-description
	|	anonymous-description
	|	set-description
	|	named-metalevel-description
	|	anonymous-metalevel-description
	|	named-default-description
	|	anonymous-default-description
single-description	::=	`(define` *name* [`:ephemeral`] feature-description*`)`
named-description	::=	`(define` (*constituent [:id] name*) [`:ephemeral`] feature-description*`)`

```
anonymous-description           ::= (define (constituent) [:ephemeral]
                                        feature-description*)

set-description                 ::= (define (set name)
                                        [name | anonymous-description]*)

named-metalevel-description     ::= (define (metalevel name) [:ephemeral]
                                        feature-description*)

anonymous-metalevel-description ::= (define (metalevel (meta+ of name))
                                        [:ephemeral]
                                        feature-description*)

named-default-description       ::= (define (default name)
                                        [name | anonymous-description]*)

anonymous-default-description   ::= (define (default (default meta* of name))
                                        [name | anonymous-description]*)

feature-description             ::= (feature-name v-expression*)
                                 |  ((feature-name v-expression+))

v-expression                    ::= name
                                 |  'symbol | string | number
                                 |  anonymous-description
                                 |  path-reference
                                 |  eval-expression
                                 |  compact-method
                                 |  reification
                                 |  metalevel-operation

query-expression                ::= path-reference
                                 |  eval-expression
                                 |  metalevel-operation

path-reference                  ::= (>> v-expression* [of v-expression])
                                 |  (!>> v-expression* [of v-expression])
                                 |  (?>> v-expression* [of v-expression])
                                 |  (*>> v-expression* [of v-expression])

eval-expression                 ::= (noos-eval [v-expression])
                                 |  (known-eval [v-expression])
                                 |  (exists-eval [v-expression])
                                 |  (all-eval [v-expression])
```

compact-method	::=	(*name* v-expression*)
reification	::=	(`reify` v-expression)
metalevel-operation	::=	(`meta` [v-expression])
	|	(`default` [v-expression])
	|	(`referent` [v-expression])
	|	(`current-task` [v-expression])
	|	(`task` v-expression [`of` v-expression])

C.1 Compact syntax for closed methods

Compact methods allows the defintion of closed methods by position instead of by name. Since definition by position assumes that required features of a built-in method are specified in a particular order, we introduce here, for each built-in method, the name of the method in compact syntax (using typewriter font) and the assumed order of parameters (inside angles). Parameters can be any v-expression such as a feature value.

When we use the compact syntax we have to specify a value for all the required features and only the required features can be specified. Otherwise an error is produced.

Compact syntax for built-in methods such as `conjunction`, `disjunction`, `union`, `intersection`, `add`, `mult`, `max`, and `min` allows the definition of n arguments. In fact, these definitions are translated as a composition of compact expressions. For instance, the following compact expression:

```
(max 1 3 5 4 2)
```

is translated to the following compact expression:

```
(max 1 (max 3 (max 5 (max 4 2))))
```

Compact syntax for Noos built-in methods is the following:

conditional	::=	(`if` <condition> <result> [<otherwise>])
not	::=	(`not` <item>)
conjunction	::=	(`and` <item1> <item2> $\cdots$ <itemn>)
disjunction	::=	(`or` <item1> <item2> $\cdots$ <itemn>)
difference	::=	(`difference` <set1> <set2>)
member	::=	(`member` <element> <set> [<test>])
union	::=	(`union` <set1> <set2> $\cdots$ <setn>)

intersection	::=	(`intersection` <set1> <set2> ⋯ <setn>)
empty-set?	::=	(`empty-set?` <set>)
cardinal	::=	(`cardinal` <set>)
add	::=	(`+` <item1> <item2> ⋯ <itemn>)
subtract	::=	(`-` <amount> <minus>)
mult	::=	(`*` <item1> <item2> ⋯ <itemn>)
div	::=	(`/` <item1> <item2>)
max	::=	(`max` <item1> <item2> ⋯ <itemn>)
min	::=	(`min` <item1> <item2> ⋯ <itemn>)
higher-than?	::=	(`>` <is-higher> <than>)
higher-equal-than?	::=	(`>=` <is-higher> <than>)
lower-than?	::=	(`<` <is-lower> <than>)
lower-equal-than?	::=	(`<=` <is-lower> <than>)
identity?	::=	(`identity?` <item1> <item2>)
subsumption	::=	(`subsumes?` <pattern> <source>)
equivalence	::=	(`equivalent?` <item1> <item2>)

Appendix D

Built-in Methods

This Appendix describes the collection of all Noos built-in methods. The goal of this Appendix is to specify the inference involved in the evaluation of each built-in method. We will not provide examples of their use in Noos. Each built-in method is described in three parts:

1. First, we will describe its required features indicating the least required sort for each feature, and the least sort of the value yielded in the evaluation of the method. We will use the following format:

   $$\textbf{method-name}[par1 \doteq s1 \ \cdots \ parn \doteq sn] \longrightarrow s$$

   where **method-name** is the name of the built-in method; $par1, \ldots, parn$ are the names of the required features for the method; $s1, \ldots, sn$ are the least required sorts of values for each required feature; and s is the least sort of the value yielded in the evaluation of the method.

   When least required sorts for feature values are not preserved, an error is produced.

2. Then, a brief description of the evaluation of the method will be provided. We will talk about feature values using their *pari* names. For instance, *item1*<*item2* means that value of feature *item1* is lower to value of feature *item2*.

3. Finally, the corresponding inference rule, using DDL notation, will be introduced.

D.1 General

Conditional

$[\mathit{condition} \doteq \mathit{boolean} \quad \mathit{result} \doteq \{\mathit{any}\} \quad \mathit{otherwise} \doteq \{\mathit{any}\}] \longrightarrow \{\mathit{any}\}$

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, and its result is the value yielded by conditional method.

Inference rules δ_m^{cond} performing the evaluation of a **conditional** method m are the following:

$$\delta_m^{cond} = \frac{\begin{array}{c} condition \doteq true \\ result \doteq c' \end{array}}{result(m) \doteq c'}, \qquad \delta_m^{cond} = \frac{\begin{array}{c} condition \doteq false \\ otherwise \doteq c' \end{array}}{result(m) \doteq c'}$$

Sequence $[] \longrightarrow \{any\}$

The **sequence** method allows the definition of a sequential chaining of subtasks (relatively to the writing order) and its result is the value yielded by the last subtask. **Sequence** has no required features but, at least, one subtask has to be defined. The names of the subtasks can be any feature name.

Since the name of the subtasks are not predetermined, given a specific **sequence** method m with subtasks $t1, \ldots, tn$, the evaluation of m is formalized as the sequence of task programs $\pi_{m.t1}; \ldots; \pi_{m.tn}$ (see Section 5.13.4) followed by an inference rule δ_m^{seq} as follows:

$$\delta_m^{seq} = \frac{tn \doteq c}{result(m) \doteq c}$$

D.2 Comparison methods

Noos provides three different comparison methods for testing equality and inclusion relationships between of two feature terms. The **identity?** method is the most specific, the **subsumption** method is the most general, and the **equivalence** method is more general than **identity?** and more specific than **subsumption**.

Identity? $[item1 \doteq any \quad item2 \doteq any] \longrightarrow boolean$

The **identity?** method compares if the values of features *item1* and *item2* are the same yielding as result *true* when are the same and *false* otherwise.

Inference rules $\delta_m^{id?}$ performing the evaluation of an **identity?** method m are the following:

$$\delta_m^{id?} = \frac{\begin{array}{c} item1 \doteq c_1 \\ item2 \doteq c_1 \end{array}}{result(m) \doteq true}, \qquad \delta_m^{id?} = \frac{\begin{array}{c} item1 \doteq c_1 \\ item2 \doteq c_2 \end{array}}{result(m) \doteq false}$$

Subsumption $[source \doteq \{any\} \quad pattern \doteq any] \longrightarrow boolean$

The **subsumption** method checks if *pattern* subsumes *source* yielding as result *true* when *pattern* subsumes *source* and *false* otherwise (see Section 5.6).

Inference rules δ_m^{subsum} performing the evaluation of a **subsumption** method m are the following:

$$\delta_m^{subsum} = \frac{\begin{array}{c} source \doteq c_1 \\ pattern \doteq c_2 \\ \text{“}c_1 \sqsubseteq c_2\text{”} \end{array}}{result(m) \doteq true}, \qquad \delta_m^{subsum} = \frac{\begin{array}{c} source \doteq c_1 \\ pattern \doteq c_2 \\ \text{“}c_1 \not\sqsubseteq c_2\text{”} \end{array}}{result(m) \doteq false}$$

Equivalence $[item1 \doteq any \quad item2 \doteq any] \longrightarrow boolean$

The **equivalence** method checks if *item1* subsumes *item2* and *item2* subsumes *item1* yielding as result *true* when this is the case and *false* otherwise.

Inference rules δ_m^{equiv} performing the evaluation of an **equivalence** method m are the following:

$$\delta_m^{equiv} = \frac{\begin{array}{c} source \doteq c_1 \\ pattern \doteq c_2 \\ \text{“}c_1 \sqsubseteq c_2 \wedge c_2 \sqsubseteq c_1\text{”} \end{array}}{result(m) \doteq true}, \qquad \delta_m^{equiv} = \frac{\begin{array}{c} source \doteq c_1 \\ pattern \doteq c_2 \\ \text{“}c_1 \not\sqsubseteq c_2 \vee c_2 \not\sqsubseteq c_1\text{”} \end{array}}{result(m) \doteq false}$$

D.3 Filter methods

Subsumption-matching $[pattern \doteq any \quad sources \doteq \{any\}] \longrightarrow any$

The **subsumption-matching** method yields an element of the set in feature *sources* such that is subsumed by *pattern*. Backtracking on a **subsumption-matching** method yields, consecutively, all other elements from *sources* also subsumed by *pattern* (see next).

Filter-by-subsumption $[pattern \doteq any \quad sources \doteq \{any\}] \longrightarrow \{any\}$

The **filter-by-subsumption** method yields the subset of elements of *sources* such that are subsumed by *pattern*.

The **filter-by-subsumption** method is defined using the **subsumption-matching** method and by refinement of the **all-eval** method as follows:

```
(define (All-Eval Filter-By-Subsumption)
  (pattern )
  (sources )
  (methods (define (Subsumption-Matching)
             (pattern (>> pattern))
             (sources (>> sources))))
```

D.4 Arithmetic methods

All the arithmetic methods require that the values of their required features be numbers. A non-number value in a feature produces an error. Moreover, each method works on all types of numbers and automatically performs any required coercions when parameters are of different types in the same way that CommonLisp.

Add $[item1 \doteq number \quad item2 \doteq number] \longrightarrow number$

The **add** method yields the sum of *item1* and *item2*.

The inference rule δ_m^{add} performing the evaluation of an **add** method m is the following:

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

Subtract $[amount \doteq number \quad minus \doteq number] \longrightarrow number$

The **subtract** method yields the subtraction of *amount* and *minus*.

The inference rule δ_m^{sub} performing the evaluation of a **subtract** method m is the following:

$$\delta_m^{sub} = \frac{\begin{array}{c} amount \doteq c_1 \\ minus \doteq c_2 \\ \text{“}c' = c_1 - c_2\text{”} \end{array}}{result(m) \doteq c'}$$

Mult $[item1 \doteq number \quad item2 \doteq number] \longrightarrow number$

The **mult** method yields the product of *item1* and *item2*.

The inference rule δ_m^{mult} performing the evaluation of a **mult** method m is the following:

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

Div $[item1 \doteq number \quad item2 \doteq number] \longrightarrow number$

The **div** method yields the division of *item1* by *item2*.

The inference rule δ_m^{div} performing the evaluation of a **div** method m is the following:

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

Max $[item1 \doteq number \quad item2 \doteq number] \longrightarrow number$

The **max** method yields the greatest number of *item1* and *item2*.

Inference rules δ_m^{max} performing the evaluation of a **max** method m are the following:

$$\delta_m^{max} = \frac{\begin{array}{c} item1 \doteq c_1 \\ item2 \doteq c_2 \\ \text{“}c_1 > c_2\text{”} \end{array}}{result(m) \doteq c_1}, \qquad \delta_m^{max} = \frac{\begin{array}{c} item1 \doteq c_1 \\ item2 \doteq c_2 \\ \text{“}c_1 \leq c_2\text{”} \end{array}}{result(m) \doteq c_2}$$

Min $[item1 \doteq number \quad item2 \doteq number] \longrightarrow number$

The **min** method yields the least number between *item1* and *item2*.

Inference rule δ_m^{min} performing the evaluation of a **min** method m are the following:

$$\delta_m^{max} = \frac{\begin{array}{c} item1 \doteq c_1 \\ item2 \doteq c_2 \\ \text{“}c_1 < c_2\text{”} \end{array}}{result(m) \doteq c_1}, \qquad \delta_m^{max} = \frac{\begin{array}{c} item1 \doteq c_1 \\ item2 \doteq c_2 \\ \text{“}c_1 \geq c_2\text{”} \end{array}}{result(m) \doteq c_2}$$

D.4.1 Numeric comparisons

Higher-than? $[\textit{is-higher} \doteq number \quad than \doteq number] \longrightarrow boolean$

The **higher-than?** method yields *true* if *is-higher* is higher than *than*. Otherwise yields *false*.

Inference rules $\delta_m^{>}$ performing the evaluation of a **higher-than?** method m are the following:

$$\delta_m^{>} = \frac{\begin{array}{c} \textit{is-higher} \doteq c_1 \\ than \doteq c_2 \\ \text{“}c_1 > c_2\text{”} \end{array}}{result(m) \doteq true}, \qquad \delta_m^{>} = \frac{\begin{array}{c} \textit{is-higher} \doteq c_1 \\ than \doteq c_2 \\ \text{“}c_1 \leq c_2\text{”} \end{array}}{result(m) \doteq false}$$

Higher-equal-than? $[\textit{is-higher} \doteq number \quad than \doteq number] \longrightarrow boolean$

The **higher-equal-than?** method yields *true* if *is-higher* is higher or equal than *than*. Otherwise yields *false*.

Inference rules $\delta_m^{\geq}$ performing the evaluation of a **higher-equal-than?** method m are the following:

$$\delta_m^{\geq} = \frac{\begin{array}{c} \textit{is-higher} \doteq c_1 \\ \textit{than} \doteq c_2 \\ \text{“}c_1 \geq c_2\text{”} \end{array}}{result(m) \doteq true}, \qquad \delta_m^{\geq} = \frac{\begin{array}{c} \textit{is-higher} \doteq c_1 \\ \textit{than} \doteq c_2 \\ \text{“}c_1 < c_2\text{”} \end{array}}{result(m) \doteq false}$$

Lower-than? [*is-lower* $\doteq$ *number* *than* $\doteq$ *number*] $\longrightarrow$ *boolean*

The **lower-than?** method yields *true* if *is-lower* is lower than *than*. Otherwise yields *false*.

Inference rules $\delta_m^{<}$ performing the evaluation of a **lower-than?** method m are the following:

$$\delta_m^{<} = \frac{\begin{array}{c} \textit{is-lower} \doteq c_1 \\ \textit{than} \doteq c_2 \\ \text{“}c_1 < c_2\text{”} \end{array}}{result(m) \doteq true}, \qquad \delta_m^{<} = \frac{\begin{array}{c} \textit{is-lower} \doteq c_1 \\ \textit{than} \doteq c_2 \\ \text{“}c_1 \geq c_2\text{”} \end{array}}{result(m) \doteq false}$$

Lower-equal-than? [*is-lower* $\doteq$ *number* *than* $\doteq$ *number*] $\longrightarrow$ *boolean*

The **lower-equal-than?** method yields *true* if *is-lower* is lower or equal than *than*. Otherwise yields *false*.

Inference rules $\delta_m^{\leq}$ performing the evaluation of a **lower-equal-than?** method m are the following:

$$\delta_m^{\leq} = \frac{\begin{array}{c} \textit{is-lower} \doteq c_1 \\ \textit{than} \doteq c_2 \\ \text{“}c_1 \leq c_2\text{”} \end{array}}{result(m) \doteq true}, \qquad \delta_m^{\leq} = \frac{\begin{array}{c} \textit{is-lower} \doteq c_1 \\ \textit{than} \doteq c_2 \\ \text{“}c_1 > c_2\text{”} \end{array}}{result(m) \doteq false}$$

D.5 Methods on sets

All the methods on sets require that the values of their required features be sets of elements. If one of them is a singleton is considered as a set with one element.

Empty-set? [*set* $\doteq$ $\{any\}$] $\longrightarrow$ *boolean*

The **empty-set?** method yields *true* if *set* is *empty-set* and *false* otherwise.

Inference rules $\delta_m^{emp?}$ performing the evaluation of an **empty-set?** method m are the following:

$$\delta_m^{emp?} = \frac{\textit{set} \doteq \textit{empty-set}}{result(m) \doteq true}, \qquad \delta_m^{emp?} = \frac{\textit{set} \doteq \{c_1 \cdots c_n\}}{result(m) \doteq false}$$

Cardinal $[set \doteq \{any\}] \longrightarrow number$

The **cardinal** method yields the number of elements in *set*.

The inference rule δ_m^{card} performing the evaluation of a **cardinal** method m is the following:

$$\delta_m^{card} = \frac{set \doteq \{c_1 \cdots c_n\}}{result(m) \doteq n}$$

Union $[set1 \doteq \{any\} \quad set2 \doteq \{any\}] \longrightarrow \{any\}$

The **union** method yields the union of sets *set1* and *set2*.

The inference rule δ_m^{union} performing the evaluation of a **union** method m is the following:

$$\delta_m^{union} = \frac{\begin{array}{c} set1 \doteq \{c_1 \cdots c_n\} \\ set2 \doteq \{c'_1 \cdots c'_m\} \\ \text{“}S = \{c_1 \cdots c_n\} \cup \{c'_1 \cdots c'_m\}\text{”} \end{array}}{result(m) \doteq S}$$

Intersection $[set1 \doteq \{any\} \quad set2 \doteq \{any\}] \longrightarrow \{any\}$

The **intersection** method yields the intersection of sets *set1* and *set2*.

The inference rule δ_m^{inter} performing the evaluation of an **intersection** method m is the following:

$$\delta_m^{inter} = \frac{\begin{array}{c} set1 \doteq \{c_1 \cdots c_n\} \\ set2 \doteq \{c'_1 \cdots c'_m\} \\ \text{“}S = \{c_1 \cdots c_n\} \cap \{c'_1 \cdots c'_m\}\text{”} \end{array}}{result(m) \doteq S}$$

Difference $[set1 \doteq \{any\} \quad set2 \doteq \{any\}] \longrightarrow \{any\}$

The **difference** method yields the set difference of *set1* and *set2*.

The inference rule δ_m^{diff} performing the evaluation of a **difference** method m is the following:

$$\delta_m^{diff} = \frac{\begin{array}{c} set1 \doteq \{c_1 \cdots c_n\} \\ set2 \doteq \{c'_1 \cdots c'_m\} \\ \text{“}S = \{c_1 \cdots c_n\} \setminus \{c'_1 \cdots c'_m\}\text{”} \end{array}}{result(m) \doteq S}$$

Member $[set \doteq \{any\} \quad item \doteq any \quad test \doteq symbol] \longrightarrow boolean$

The **member** method yields *true* if *item* is found in *set* using the comparison criterion *test*. Otherwise yields *false*. Three comparison criteria can be used: *identity*, *subsumption*, and *equivalence* corresponding to the comparison methods described before (see Section D.2).

Inference rules δ_m^{member} performing the evaluation of a **member** method m with *identity* comparison criterion are the following:

$$\delta_m^{member} = \frac{\begin{array}{c} set \doteq \{c_1 \cdots c_n\} \\ element \doteq c_i \\ test \doteq \text{'}identity \\ \text{“}c_i \in \{c_1 \cdots c_n\}\text{”} \end{array}}{result(m) \doteq true} \delta_m^{member} = \frac{\begin{array}{c} set \doteq \{c_1 \cdots c_n\} \\ element \doteq c_i \\ test \doteq \text{'}identity \\ \text{“}c_i \notin \{c_1 \cdots c_n\}\text{”} \end{array}}{result(m) \doteq true}$$

Inference rules δ_m^{member} performing the evaluation of a **member** method m with *subsumption* comparison criterion are the following:

$$\delta_m^{member} = \frac{\begin{array}{c} set \doteq S \\ element \doteq c_i \\ test \doteq \text{'}subsumption \\ \text{“}\exists c_j \in S : c_i \sqsubseteq c_j\text{”} \end{array}}{result(m) \doteq true}, \qquad \delta_m^{member} = \frac{\begin{array}{c} set \doteq S \\ element \doteq c_i \\ test \doteq \text{'}subsumption \\ \text{“}\nexists c_j \in S : c_i \sqsubseteq c_j\text{”} \end{array}}{result(m) \doteq true}$$

Inference rules δ_m^{member} performing the evaluation of a **member** method m with *equivalence* comparison criterion are the following:

$$\delta_m^{member} = \frac{\begin{array}{c} set \doteq S \\ element \doteq c_i \\ test \doteq \text{'}equivalence \\ \text{“}\exists c_j \in S : c_j \sqsubseteq c_i \sqsubseteq c_j\text{”} \end{array}}{result(m) \doteq true}, \delta_m^{member} = \frac{\begin{array}{c} set \doteq S \\ element \doteq c_i \\ test \doteq \text{'}equivalence \\ \text{“}\nexists c_j \in S : c_j \sqsubseteq c_i \sqsubseteq c_j\text{”} \end{array}}{result(m) \doteq true}$$

D.6 Logic methods

All the logic methods require that the values of their required features be boolean. A non-boolean value in a feature produces an error.

Not $[item \doteq boolean] \longrightarrow boolean$

The **not** method yields the negation of *item*.

Inference rules δ_m^{not} performing the evaluation of a **not** method m are the following:

$$\delta_m^{not} = \frac{item \doteq false}{result(m) \doteq true}, \qquad \delta_m^{not} = \frac{item \doteq true}{result(m) \doteq false}$$

Conjunction $[item1 \doteq boolean \quad item2 \doteq boolean] \longrightarrow boolean$

The **conjunction** method yields *true* if *item1* and *item2* are both *true* and *false* otherwise.

The inference rule δ_m^{conj} performing the evaluation of a **conjunction** method m is the following:

$$\delta_m^{conj} = \frac{\begin{array}{c} item1 \doteq b_1 \\ item2 \doteq b_2 \\ \text{“}b = b_1 \wedge b_2\text{”} \end{array}}{result(m) \doteq b}$$

Disjunction $[item1 \doteq boolean \quad item2 \doteq boolean] \longrightarrow boolean$

The **disjunction** method yields *true* if either *item1* or *item2* is *true* and *false* if both are *false*.

The inference rule δ_m^{disj} performing the evaluation of a **disjunction** method m is the following:

$$\delta_m^{disj} = \frac{\begin{array}{c} item1 \doteq false \\ item2 \doteq false \\ \text{“}b = b_1 \vee b_2\text{”} \end{array}}{result(m) \doteq b}$$

D.7 Retrieval methods

Retrieval methods provide a powerful mechanism for accessing to the episodic memory contents. There are three built-in methods for retrieval defined in Noos: **retrieve-by-pattern**, **retrieve-by-task**, and **retrieve-by-feature-value**.

Retrieve-by-pattern $[pattern \doteq any] \longrightarrow \{any\}$

The **retrieve-by-pattern** method yields the collection of terms, from the episodic memory, subsuming the *pattern*.

The inference rule δ_m^{rbp} performing the evaluation of a **retrieve-by-pattern** method m on the episodic memory U is the following:

$$\delta_m^{rbp} = \frac{\begin{array}{c} pattern \doteq c \\ S = \{c_i \in U | c \sqsubseteq c_i\} \end{array}}{result(m) \doteq S}$$

Retrieve-by-task $[task\text{-}name \doteq symbol] \longrightarrow \{any\}$

The **retrieve-by-task** method yields the collection of terms, from the episodic memory, that have solved the task *task-name*.

The inference rule δ_m^{rbt} performing the evaluation of a **retrieve-by-task** method m on the episodic memory U is the following:

$$\delta_m^{rbt} = \frac{\begin{array}{c} \textit{task-name} \doteq \textit{'f} \\ S = \{c_i \in U | c_i.f \text{ is defined}\} \end{array}}{result(m) \doteq S}$$

Retrieve-by-feature-value [*task-name* $\doteq$ *symbol* *value* $\doteq$ *any*] $\longrightarrow$ $\{any\}$

The **retrieve-by-feature-value** method yields the collection of terms, from the episodic memory, that have *value* in *task-name*.

The inference rule δ_m^{rbfv} performing the evaluation of a **retrieve-by-feature-value** method m on the episodic memory U is the following:

$$\delta_m^{rbfv} = \frac{\begin{array}{c} \textit{task-name} \doteq \textit{'f} \\ value \doteq c \\ S = \{c_i \in U | c_i.f \doteq c\} \end{array}}{result(m) \doteq S}$$

D.8 Preferences

We have described preferences in Section 3.4. Now we will briefly describe preference methods indicating their required features and their corresponding inference rules using the definitions of preference operations given in Section 5.10. The reader can consult Section 3.4 for examples of the use of preferences in Noos.

We have explained that a preference method takes a set of source elements and builds a preference taking into account an ordering criterion. In fact, built-in preference methods are more powerful: when the set of source elements is already a partially ordered set, the new preference is added using the hierarchical union operator. Since all preference methods performs a hierarchical union, for the sake of clarity we will first describe a preference method without the hierarchical union and then, we will present its inference rule incorporating the hierarchical union.

Increasing-order-preference [*poset* $\doteq$ *poset* *feature* $\doteq$ *symbol*] $\longrightarrow$ *poset*

The *increasing-order-preference* method takes the set of elements in *poset*, the feature name *feature* of a feature with numeric value, and yields a new preference where the most preferred elements are those with a higher value in the specified feature. If any value for *feature* is not numeric an error is produced.

The inference rule δ_m^{incp} performing the evaluation of a **increasing-order-preference** method m is the following:

$$\delta_m^{incp} = \frac{\begin{array}{c} poset \doteq \langle S, \prec \rangle \\ \textit{feature} \doteq \textit{'f'} \\ \text{``} \prec' = \{(c_i, c_j) | c_i, c_j \in S, c_i.f > c_j.f\} \\ \text{``}p' = \langle S, \prec \rangle \bullet \langle S, \prec' \rangle\text{''} \end{array}}{result(m) \doteq p'}$$

Decreasing-order-preference $[poset \doteq poset \quad feature \doteq symbol] \longrightarrow poset$

The *decreasing-order-preference* method takes the set of elements in *poset*, the feature name *feature* of a feature with numeric value, and yields a new preference where the most preferred elements are those with a lesser value in the specified feature. If any value for *feature* is not numeric an error is produced.

The inference rule δ_m^{decp} performing the evaluation of a **decreasing-order-preference** method m is the following:

$$\delta_m^{decp} = \frac{\begin{array}{c} poset \doteq \langle S, \prec \rangle \\ feature \doteq \text{'}f \\ \text{“} \prec' = \{(c_i, c_j) | c_i, c_j \in S, c_i.f < c_j.f\}\text{”} \\ \text{“}p' = \langle S, \prec \rangle \bullet \langle S, \prec' \rangle\text{”} \end{array}}{result(m) \doteq p'}$$

Higher-threshold-preference
$[poset \doteq poset \quad feature \doteq symbol \quad threshold \doteq number] \longrightarrow poset$

The *higher-threshold-preference* method takes the set of elements in *poset*, the feature name *feature* of a feature with numeric value, a *threshold*, and yields a new preference where the elements with a higher value than *value* in *feature* are preferred to elements with lower or equal value than *value*. If any value for *feature* is not numeric an error is produced.

The inference rule δ_m^{htp} performing the evaluation of a **higher-threshold-preference** method m is the following:

$$\delta_m^{htp} = \frac{\begin{array}{c} poset \doteq \langle S, \prec \rangle \\ feature \doteq \text{'}f \\ threshold \doteq c \\ \text{“} \prec' = \{(c_i, c_j) | c_i, c_j \in S, c_i.f > c, c_j.f \leq c\}\text{”} \\ \text{“}p' = \langle S, \prec \rangle \bullet \langle S, \prec' \rangle\text{”} \end{array}}{result(m) \doteq p'}$$

Lower-threshold-preference
$[poset \doteq poset \quad feature \doteq symbol \quad threshold \doteq number] \longrightarrow poset$

The *lower-threshold-preference* method takes the set of elements in *poset*, the feature name *feature* of a feature with numeric value, a *threshold*, and yields a new preference where the elements with a lower value than *value* in *feature* are preferred to elements with higher or equal value than *value*. If any value for *feature* is not numeric an error is produced.

The inference rule δ_m^{ltp} performing the evaluation of a **lower-threshold-preference** method m is the following:

$$\delta_m^{ltp} = \frac{\begin{array}{c} poset \doteq \langle S, \prec \rangle \\ feature \doteq \text{'}f \\ threshold \doteq c \\ \text{``} \prec' = \{(c_i, c_j) | c_i, c_j \in S, c_i.f < c, c_j.f \geq c\}\text{''} \\ \text{``}p' = \langle S, \prec \rangle \bullet \langle S, \prec' \rangle\text{''} \end{array}}{result(m) \doteq p'}$$

Subsumption-preference $[poset \doteq poset \quad pattern \doteq any] \longrightarrow poset$

The *subsumption-preference* method takes the set of elements in *poset*, the term *pattern*, and yields a new preference where the preferred elements are those *pattern* subsumes.

The inference rule δ_m^{subp} performing the evaluation of a **subsumption-preference** method m is the following:

$$\delta_m^{subp} = \frac{\begin{array}{c} poset \doteq \langle S, \prec \rangle \\ pattern \doteq c \\ \text{``} \prec' = \{(c_i, c_j) | c_i, c_j \in S, c_i.f \sqsubseteq c, c_j.f \not\sqsubseteq c\}\text{''} \\ \text{``}p' = \langle S, \prec \rangle \bullet \langle S, \prec' \rangle\text{''} \end{array}}{result(m) \doteq p'}$$

Equal-value-preference
$[poset \doteq poset \quad feature \doteq symbol \quad value \doteq number] \longrightarrow poset$

The *equal-value-preference* method takes the set of elements in *poset*, the feature name *feature* of a feature with numeric value, a *value*, and yields a new preference where the elements with *value* in *feature* are preferred to others.

The inference rule δ_m^{evp} performing the evaluation of a **equal-value-preference** method m is the following:

$$\delta_m^{evp} = \frac{\begin{array}{c} poset \doteq \langle S, \prec \rangle \\ feature \doteq \text{'}f \\ value \doteq c \\ \text{``} \prec' = \{(c_i, c_j) | c_i, c_j \in S, c_i.f = c, c_j.f \neq c\}\text{''} \\ \text{``}p' = \langle S, \prec \rangle \bullet \langle S, \prec' \rangle\text{''} \end{array}}{result(m) \doteq p'}$$

User-preference $[poset \doteq poset] \longrightarrow poset$

The *user-preference* method takes the set of elements in *poset* and yields a new preference according to the answer provided by the user using a window interface.

Inversion $[poset \doteq poset] \longrightarrow poset$

The *inversion* method yields a new preference, inversion of preference *poset*.

The inference rule δ_m^{inver} performing the evaluation of a **inversion** method m is the following:

$$\delta_m^{inver} = \frac{\begin{array}{c} poset \doteq p \\ \text{“}p' = p^{-1}\text{”} \end{array}}{result(m) \doteq p'}$$

T-intersection $[poset1 \doteq poset \quad poset2 \doteq poset] \longrightarrow poset$

The *t-intersection* method yields a new preference by restricting preferences *poset1* and *poset2* to the elements of their intersection and, then, performing a transitive union on the resulting preferences (see Definition 5.25 in Section 5.10).

The inference rule $\delta_m^{\overline{\cap}}$ performing the evaluation of a *t-intersection* method m is the following:

$$\delta_m^{\overline{\cap}} = \frac{\begin{array}{c} poset1 \doteq p_1 \\ poset2 \doteq p_2 \\ \text{“}p = p_1 \overline{\cap} p_2\text{”} \end{array}}{result(m) \doteq p}$$

C-intersection $[poset1 \doteq poset \quad poset2 \doteq poset] \longrightarrow poset$

The *c-intersection* method yields a new preference by restricting preferences *poset1* and *poset2* to the elements of their intersection and, then, performing an intersection on the resulting preferences (see Definition 5.26 in Section 5.10).

The inference rule $\delta_m^{\underline{\cap}}$ performing the evaluation of a *c-intersection* method m is the following:

$$\delta_m^{\underline{\cap}} = \frac{\begin{array}{c} poset1 \doteq p_1 \\ poset2 \doteq p_2 \\ \text{“}p = p_1 \underline{\cap} p_2\text{”} \end{array}}{result(m) \doteq p}$$

T-union $[poset1 \doteq poset \quad poset2 \doteq poset] \longrightarrow poset$

The *t-union* method yields a new preference by extending preferences *poset1* and *poset2* to the elements of their union and, then, performing a transitive union on the resulting preferences (see Definition 5.27 in Section 5.10).

The inference rule $\delta_m^{\overline{\uplus}}$ performing the evaluation of a *t-union* method m is the following:

$$\delta_m^{\overline{\uplus}} = \frac{\begin{array}{c} poset1 \doteq p_1 \\ poset2 \doteq p_2 \\ \text{``}p = p_1 \overline{\uplus} p_2\text{''} \end{array}}{result(m) \doteq p}$$

C-union $[poset1 \doteq poset \quad poset2 \doteq poset] \longrightarrow poset$

The *c-union* method yields a new preference by extending preferences *poset1* and *poset2* to the elements of their union and, then, performing an intersection on the resulting preferences (see Definition 5.28 in Section 5.10).

The inference rule $\delta_m^{\underline{\uplus}}$ performing the evaluation of a *c-union* method m is the following:

$$\delta_m^{\underline{\uplus}} = \frac{\begin{array}{c} poset1 \doteq p_1 \\ poset2 \doteq p_2 \\ \text{``}p = p_1 \underline{\uplus} p_2\text{''} \end{array}}{result(m) \doteq p}$$

H-union $[higher - poset \doteq poset \quad lower - poset \doteq poset] \longrightarrow poset$

The *h-union* method yields a new preference by extending preferences *higher-poset* and *lower-poset* to the elements of their union and, then, performing a hierarchical union on the resulting preferences (see Definition 5.30 in Section 5.10).

The inference rule $\delta_m^{\bullet}$ performing the evaluation of a *h-union* method m is the following:

$$\delta_m^{\bullet} = \frac{\begin{array}{c} higher - poset \doteq p_1 \\ poset2 \doteq p_2 \\ \text{``}p = p_1 \bullet p_2\text{''} \end{array}}{result(m) \doteq p}$$

D.9 Methods for interaction

These kind of methods allow user interaction within the inference process. These methods are commonly used for dynamically choosing among alternatives and for introducing new values for features.

We will describe the two built-in methods provided by Noos without introducing inference rules.

Ask-option-to-user $[question \doteq string \quad options \doteq set] \longrightarrow set$

The **ask-option-to-user** method shows *question* and *options* to the user and yields the subset of elements of *options* chosed by the user.

Ask-value-to-user [$question \doteq string$] $\longrightarrow$ {any}

The **ask-value-to-user** method shows *question* and request a value to the user. The value can be any feature term already defined. Otherwise an error is produced.

D.10 Query methods

We have described query-methods in Section 3.3.6. Now we will briefly describe query-methods indicating their required features. Inference rules and DDL programs formalizing inference of query-methods are given in Section 5.13. The reader can consult Section 3.3.6 for examples of the use of query-methods in Noos.

Infer-value [$feature \doteq symbol \quad domain \doteq \{any\}$] $\longrightarrow$ {any}

The **infer-value** method takes a feature name f in *feature*; a unit or a set of units S in *domain*; performs tasks $f(s_i)$ for all units in S; and yields as result the union of the values inferred in those tasks.

Known-value [$feature \doteq symbol \quad domain \doteq \{any\}$] $\longrightarrow$ {any}

The **known-value** method takes a feature name f in *feature*; a unit or a set of units S in *domain*; and determines if a solution to a task f for the set of units in S have already been computed, yielding a boolean accordingly.

Exists-value [$feature \doteq symbol \quad domain \doteq \{any\}$] $\longrightarrow$ {any}

The **exists-value** method takes a feature name f in *feature*; a unit or a set of units S in *domain*; and determines if any solution to a task f for the set of units in S exists, yielding a boolean accordingly.

All-values [$feature \doteq symbol \quad domain \doteq \{any\}$] $\longrightarrow$ {any}

The **all-values** method takes a feature name f in *feature*; a unit or a set of units S in *domain*; determines the set of all inferrable values of tasks $f(s_i)$ for all units in S; and yields as result the union of all those values.

D.11 Eval methods

Eval-methods have been described in Section 3.3.6. Now we will briefly describe eval-methods indicating their required features. Inference rules and DDL programs formalizing inference of eval-methods are given in Section 5.13.

Noos-eval [$methods \doteq \{method\}$] $\longrightarrow$ {any}

The **noos-eval** method takes a method or a set of methods M in *methods*; performs their evaluation; and yields as result the union of the values inferred in the evaluation of those methods.

Known-eval $[methods \doteq \{method\}] \longrightarrow \{any\}$

The **known-eval** method takes a method or a set of methods M in *methods* and determines if a solution for methods in M have already been computed, yielding a boolean accordingly.

Exists-eval $[methods \doteq \{method\}] \longrightarrow \{any\}$

The **exists-eval** method takes a method or a set of methods M in *methods* and determines if exists a solution for methods in M, yielding a boolean accordingly.

All-eval $[methods \doteq \{method\}] \longrightarrow \{any\}$

The **all-eval** method takes a method or a set of methods M in *methods*; determines the set of all inferrable values in their evaluations; and yields as result the union of all those values.

D.12 Reflective operations

We have described reflective operations in Section 3.3.5. Reflective operations are also reified as methods. Now we will briefly describe these methods indicating their required features and no inference rules will be provided. The reader can consult Section 3.3.5 for examples of the use of reflective operations.

Meta $[object \doteq any] \longrightarrow any$

The **meta** method performs the subtask *object* and then, yields the metalevel of *object* as result.

Default $[object \doteq any] \longrightarrow any$

The **default** method performs the subtask *object* and then, yields the default metalevel of *object* as result.

Referent $[object \doteq any] \longrightarrow any$

The **referent** method performs the subtask *object* and then, yields the referent of *object* as result.

Constituent $[object \doteq any] \longrightarrow any$

The **constituent** method performs the subtask *object* and then, yields the constituent of *object* as result.

Task $[task\text{-}name \doteq symbol \quad domain \doteq any] \longrightarrow task$

The **task** method performs subtasks *task-name* and *domain* obtaining respectively a task name `F` and a term `D`. Then, yields the task term that reifies the inference `F(D)` as result.

Current-task $[method \doteq method] \longrightarrow task$

The **current-task** method performs the subtask *method*, obtaining a method `M`, and then yields as result the task term that reifies the task `M` is solving.

Bibliography

[Aamodt, 1991] Aamodt, A. (1991). *A Knowledge-Intensive, Integrated Approach to Problem Solving and Sustained Learning.* PhD thesis, University of Trondheim.

[Aamodt and Plaza, 1994] Aamodt, A. and Plaza, E. (1994). Case-based reasoning: Foundational issues, methodological variations, and system approaches. *Artificial Intelligence Communications*, 7(1):39–59. online at <url:http://www.iiia.csic.es/People/enric/AICom_ToC.html>.

[Aït-Kaci and Podelski, 1993] Aït-Kaci, H. and Podelski, A. (1993). Towards a meaning of LIFE. *J. Logic Programming*, 16:195–234.

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

[Angele et al., 1994] Angele, J., Fensel, D., and Studer, R. (1994). The model of expertise in KARL. In *Proceedings of the 2nd world congress on Expert Systems.*

[Arcos, 1995] Arcos, J. L. (1995). Configuració de problemes de planificació en Noos. In *Trobada de Joves Investigadors.*

[Arcos, 1996] Arcos, J. L. (1996). Saxex: un sistema de raonament basat en casos per a l'expressivitat musical. Master's thesis, Institut Universitari de l'Audiovisual. Universitat Pompeu Fabra.

[Arcos and López de Mántaras, 1997] Arcos, J. L. and López de Mántaras, R. (1997). Perspectives: a declarative bias mechanism for case retrieval. In Leake, D. and Plaza, E., editors, *Case-Based Reasoning. Research and Development*, number 1266 in Lecture Notes in Artificial Intelligence, pages 279–290. Springer-Verlag.

[Arcos et al., 1997a] Arcos, J. L., López de Mántaras, R., and Serra, X. (1997a). Generating expressive musical performances with Saxex. In *International workshop Kansei Technology of Emotion (AIMI'97).*

[Arcos et al., 1997b] Arcos, J. L., López de Mántaras, R., and Serra, X. (1997b). Saxex : a case-based reasoning system for generating expressive musical performances. In *International Computer Music Conference (ICMC'97)*.

[Arcos and Plaza, 1993] Arcos, J. L. and Plaza, E. (1993). A reflective architecture for integrated memory-based learning and reasoning. In Wess, S., Althoff, K., and Richter, M., editors, *Topics in Case-Based Reasoning*, number 837 in Lecture Notes in Artificial Intelligence, pages 289–300. Springer-Verlag.

[Arcos and Plaza, 1994] Arcos, J. L. and Plaza, E. (1994). Integration of learning into a knowledge modelling framework. In Steels, L., Schreiber, G., and Van de Velde, W., editors, *A Future for Knowledge Acquisition*, number 867 in Lecture Notes in Artificial Intelligence, pages 355–373. Springer-Verlag.

[Arcos and Plaza, 1995] Arcos, J. L. and Plaza, E. (1995). Reflection in Noos: An object-centered representation language for knowledge modelling. In *IJCAI-95 Workshop on Reflection and Meta-Level Architectures and their Applications in AI*.

[Arcos and Plaza, 1996] Arcos, J. L. and Plaza, E. (1996). Inference and reflection in the object-centered representation language Noos. *Journal of Future Generation Computer Systems*, 12:173–188.

[Arcos and Plaza, 1997] Arcos, J. L. and Plaza, E. (1997). Noos: an integrated framework for problem solving and learning. In *Knowledge Engineering: Methods and Languages*.

[Armengol, 1997] Armengol, E. (1997). *A Framework for Integrating Learning and Problem Solving*. PhD thesis, Universitat Politècnica de Catalunya.

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

[Armengol and Plaza, 1997] Armengol, E. and Plaza, E. (1997). Induction of feature terms with INDIE. In van Someren, M. and Widmer, G., editors, *Machine Learning: ECML-97*, Lecture Notes in Artificial Intelligence. Springer-Verlag.

[Backofen and Smolka, 1995] Backofen, R. and Smolka, G. (1995). A complete and recursive feature theory. *Theoretical Computer Science*, 146(1–2):243–268.

[Benjamins, 1993] Benjamins, R. (1993). *Problem Solving Methods for Diagnosis*. PhD thesis, University of Amsterdam.

[Benjamins, 1994] Benjamins, R. (1994). On a role of problem solving methods in knowledge acquisition - experiments with diagnostic strategies. In Steels, L., Schreiber, G., and Van de Velde, W., editors, *A Future for Knowledge Acquisition*, number 867 in Lecture Notes in Artificial Intelligence. Springer-Verlag.

[Brachman et al., 1991] Brachman, R. J., McGuinness, D. L., Patel-Schneider, P. F., Resnick, L. A., and Borgida, A. (1991). Living with CLASSIC: when and how to use a KL-ONE-like language. In Sowa, J., editor, *Principles of Semantic Networks: Explorations in the representation of knowledge*, pages 401–456. Morgan-Kaufmann.

[Cabré, 1996] Cabré, M. (1996). SHAM. sistema harmonitzador automàtic de melodies. Master's thesis, Facultat de Ciències, secció d'Enginyeria Informàtica. Universitat Autònoma de Barcelona.

[Carbonell, 1986] Carbonell, J. (1986). Derivational analogy: A theory of reconstructive problem solving and expertise acquisition. In Michalski, R. S., Carbonell, J. G., and Mitchell, T. M., editors, *Machine Learning*, volume 2, pages 371–392. Morgan Kaufmann.

[Carbonell et al., 1995] Carbonell, J., Etzioni, O., Gil, Y., Joseph, R., Knoblock, C., Minton, S., and Veloso, M. (1995). Planning and learning in PRODIGY: Overview of an integrated architecture. In Ram, A. and Leake, D. B., editors, *Goal-driven Learning*. MIT press.

[Carbonell et al., 1991] Carbonell, J., Knoblock, C. A., and Minton, S. (1991). Prodigy: An integrated architecture for planning and learning. In VanLehn, K., editor, *Architectures for Intelligence*. Lawrence Erlaum Associates.

[Carbonell, 1989] Carbonell, J. G. (1989). Special issue on machine learning. *Artificial intelligence*, 1–3(40).

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

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

[Chandrasekaran, 1986] Chandrasekaran, B. (1986). Generic tasks in knowledge-based reasoning: High-level building blocks for expert system design. *IIIE expert*, 1:23–30.

[Chandrasekaran, 1989] Chandrasekaran, B. (1989). Task structures, knowledge acquisition and machine learning. *Machine Learning*, 2:341–347.

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

[de Bruijn, 1972] de Bruijn, N. (1972). Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation. *Indag. Mat.*, 34:381–392.

[Digitool, 1996] Digitool (1996). *Macintosh Common Lisp Reference*. Digitool.

[Domingo, 1995] Domingo, M. (1995). *An Expert System Architecture for Identification in Biology*, volume 4 of *Monografies del IIIA*. IIIA–CSIC.

[Etzioni, 1988] Etzioni, O. (1988). Hypothesis filtering: a practical approach to reliable learning. In *Proceedings of Fifth International Conference on Machine Learning*, pages 416–429. San Mateo, CA: Morgan-Kaufmann.

[Fensel, 1995a] Fensel, D. (1995a). Formal specification languages in knowledge and software engineering. *The Knowledge Engineering Review*, 10(4):361–404.

[Fensel, 1995b] Fensel, D. (1995b). *The Knowledge Acquisition and Representation Language KARL*. Kluwer.

[Fensel and Benjamins, 1996] Fensel, D. and Benjamins, R. (1996). Assumptions in model-based diagnosis. In Gaines, B. and Musen, M., editors, *Proceedings of the 10th Banff Knowledge Acquisition for Knowledge Based Systems Workshop*. SRDG Publications, University of Calgary.

[Gennari et al., 1994] Gennari, J. H., Tu, S. W., Rothenfluh, T. E., and Musen, M. A. (1994). Mapping domains to methods in support of reuse. *International Journal of Human-Computer Studies*, 41:399–424.

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

[Greiner and Lenat, 1980] Greiner, R. and Lenat, D. (1980). RLL-1: A representation language language. Technical Report HPP-80-9, Comp. Science dept., Stanford University. Expanded version of the same paper in Proc. First AAAI Conference.

[Haase, 1987] Haase, K. (1987). Why representation languages are no good. Technical Report 943, MIT AI Memo.

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

[Jonckers et al., 1992] Jonckers, V., Geldof, S., and Devroede, K. (1992). the COMMET methodology and workbench in practice. In *Proceedings of the 5th International Symposium on Artificial Intelligence*, pages 341–348.

[Kamp, 1997] Kamp, G. (1997). On the admissibility of concrete domains for CBR based on description logics. In Leake, D. and Plaza, E., editors, *Case-Based Reasoning. Research and Development*, number 1266 in Lecture Notes in Artificial Intelligence, pages 223–234. Springer-Verlag.

[Karbach et al., 1991] Karbach, W., Voss, A., Schukey, R., and Drouwen, U. (1991). Model-K: Prototyping at the knowledge level. In *Proceedings Expert systems-91*, pages 501–512.

[Kiczales et al., 1991] Kiczales, G., Des Rivières, J., and Bobrow, D. G. (1991). *The Art of the Metaobject Protocol*. The MIT Press: Cambridge.

[Kolodner and Riesbek, 1986] Kolodner, J. L. and Riesbek, C. K., editors (1986). *Experience, Memory, and Reasoning*. Lawrence Erlbaum Associates.

[Koton, 1989] Koton (1989). Using experience in learning and problem solving. Technical Report 90/2, MIT/LCS.

[Laird et al., 1987] Laird, J. E., Newell, A., and Rosenbloom, P. S. (1987). Soar: an architecture for general intelligence. *Artificial Intelligence*, 33:1–64.

[Laird et al., 1986] Laird, J. E., Rosenbloom, P. S., and Newell, A. (1986). *Universal Subgoaling and Chunking*. Kluwer Academic Publishers.

[Langley, 1989] Langley, P. (1989). Toward a unified science of machine learning. *Machine Learning*, 11(2–3):111–152.

[Lavrač and Džeroski, 1994] Lavrač, N. and Džeroski, S. (1994). *Inductive Logic Programming. Techniques and Applications*. Ellis Horwood.

[Lenat, 1983] Lenat, D. B. (1983). Eurisko: A program which learns new heuristics and domain concepts. *Artificial Intelligence*, 21.

[Lerdahl and Jackendoff, 1993] Lerdahl, F. and Jackendoff, R. (1993). An overview of hierarchical structure in music. In Schwanaver, S. M. and Levitt, D. A., editors, *Machine Models of Music*. The MIT Press. Reproduced from Music Perception.

[Levy and Rousset, 1996] Levy, A. Y. and Rousset, M.-C. (1996). CARIN: A representation language combining horn rules and description logics. In Wahlster, W., editor, *Proceedings of the 12th European Conference on AI (ECAI-96)*, pages 323–327.

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

[López de Mántaras, 1991] López de Mántaras, R. (1991). A distance-based attribute selection measure for decision tree induction. *Machine Learning*, 6:81–92.

[MacGregor, 1991] MacGregor, R. M. (1991). Using a description classifier to enhace deductive inference. In *Proceedings Seventh IEEE-91 Conference on AI Applications*.

[MacGregor, 1994] MacGregor, R. M. (1994). A description classifier for the predicate calculus. In *Proceedings AAAI-94 National Conference*.

[Maes, 1988] Maes, P. (1988). Issues in computational reflection. In Maes, P. and Nardi, D., editors, *Meta-level architectures and reflection*. North-Holland.

[Martín, 1996] Martín, F. J. (1996). NoosWeb: una interficie en la World-Wide Web para Noos. Master's thesis, Facultat d'Informàtica. Universitat Politècnica de València.

[McDermott, 1988] McDermott, J. (1988). A taxonomy of problem-solving methods. In Marcus, S., editor, *Automating Knowledge Acquisition*. Kluwer.

[Michalski, 1993] Michalski, R. (1993). Inferential theory of learning as a conceptual basis for multistrategy learning. *Machine Learning*, 11(2–3):111–152.

[Minton, 1990] Minton, S. (1990). Qualitative results concerning the utility of explanation-based learning. *Artificial Intelligence*, 42:363–391.

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

[Moreno et al., 1994] Moreno, A. et al. (1994). *Aprendizaje Automático*. Edicions UPC.

[Muggleton, 1992] Muggleton, S., editor (1992). *Inductive Logic Programming*. Academic Press.

[Muggleton and De Raedt, 1994] Muggleton, S. and De Raedt, L. (1994). Inductive logic programming: Theory and methods. *Journal of Logic Programming*, 19–20:629–679.

[Narmour, 1990] Narmour, E. (1990). *The Analysis and cognition of basic melodic structures : the implication-realization model*. University of Chicago Press.

[Nebel, 1990] Nebel, B. (1990). *Reasoning and Revision in Hybrid Representation Systems*, volume 422 of *Lecture Notes in Artificial Intelligence*. Springer Verlag.

[Newell, 1982] Newell, A. (1982). The knowledge level. *Artificial Intelligence*, 18:87–127.

[Newell, 1990] Newell, A. (1990). *Unified Theories of Cognition*. Cambridge MA: Harvard University Press.

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

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

[Plaza et al., 1993] Plaza, E., Aamodt, A., Ram, A., Van de Velde, W., and van Someren, M. (1993). Integrated learning architectures. In Brazdil, P. V., editor, *Machine Learning: ECML-93*, number 667 in Lecture Notes in Artificial Intelligence, pages 429–441. Springer-Verlag.

[Plaza and Arcos, 1993a] Plaza, E. and Arcos, J. L. (1993a). Explicit inference methods in Noos. Technical Report 15, IIIA.

[Plaza and Arcos, 1993b] Plaza, E. and Arcos, J. L. (1993b). Flexible integration of multiple learning methods into a problem solving architecture. Technical Report 16, IIIA. extended version of the paper presented at ECML-94.

[Plaza and Arcos, 1993c] Plaza, E. and Arcos, J. L. (1993c). Reflection and analogy in memory-based learning. In *Multistrategy Learning Workshop MSL-93*.

[Plaza and Arcos, 1993d] Plaza, E. and Arcos, J. L. (1993d). Reflection memory and learning. Technical Report 2, IIIA.

[Plaza and Arcos, 1993e] Plaza, E. and Arcos, J. L. (1993e). Using reflection principles in the integration of learning and problem solving. Technical Report 11, IIIA. extended version of the paper presented at Integrated Learning Architectures Workshop ECML-93.

[Plaza and Arcos, 1993f] Plaza, E. and Arcos, J. L. (1993f). Using reflection principles in the integration of learning and problem solving. In *Integrated Learning Architectures Workshop ECML-93*.

[Plaza and Arcos, 1994] Plaza, E. and Arcos, J. L. (1994). Flexible integration of multiple learning methods into a problem solving architecture. In Bergadano, F. and de Raedt, L., editors, *Machine Learning: ECML-94*, number 784 in Lecture Notes in Artificial Intelligence, pages 403–406. Springer-Verlag.

[Plaza et al., 1996a] Plaza, E., Arcos, J. L., and Martín, F. (1996a). Cooperation modes among case-based reasoning agents. In *Proc. ECAI'96 Workshop on Learning in Distributed Artificial Intelligence Systems*.

[Plaza et al., 1997] Plaza, E., Arcos, J. L., and Martín, F. (1997). Cooperative case-based reasoning. In Weiss, G., editor, *Distributed Artificial Intelligence Meets Machine Learning. Learning in Multi-Agent Environments*, number 1221 in Lecture Notes in Artificial Intelligence, pages 180–201. Springer-Verlag.

[Plaza et al., 1996b] Plaza, E., López de Mántaras, R., and Armengol, E. (1996b). On the importance of similitude: An entropy-based assessment. In Smith, I. and Saltings, B., editors, *Advances in Case-based reasoning. Third European Workshop EWCBR-96*, number 1168 in Lecture Notes in Artificial Intelligence, pages 324–338. Springer-Verlag. Longer version is available as Research report IIIA-RR-96-14 online at <http://www.iiia.csic.es/Reports/1996/IIIA-RR-96.html>.

[Puerta et al., 1992] Puerta, A., Egar, J., Tu, S. W., and Musen, M. A. (1992). A multiple-method knowledge acquisition shell for the automatic generation of knowledge acquisition tools. *Knowledge Acquisition*, 4:171–196.

[Puyol-Gruart, 1995] Puyol-Gruart, J. (1995). *MILORD II: A Language for Knowledge–Based Systems*, volume 1 of *Monografies del IIIA*. IIIA–CSIC.

[Quinlan, 1990] Quinlan, J. R. (1990). Learning logical definitions from relations. *Machine Learning*, 5:239–266.

[Rademakers, 1988] Rademakers, P. (1988). Implementing reflective architectures in object-oriented languages. Technical report, Artificial Intelligence Laboratory. Vrije Universiteit Brussel. AI MEMO 88-14.

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

[Rosenbloom et al., 1991] Rosenbloom, P. S., Newell, A., and Laird, J. E. (1991). Toward the knowledge level in Soar: The role of the architecture in the use of knowledge. In VanLehn, K., editor, *Architectures for Intelligence*, pages 75–111. Lawrence Erlaum Associates.

[Russell, 1990] Russell, S. (1990). *The use of knowledge in Analogy and Induction.* Morgan Kaufmann.

[Sabater, 1997] Sabater, J. (1997). GYMEL. Sistema d'harmonització de melodies utilitzant raonament basat en casos. Master's thesis, Facultat de Ciències, secció d'Enginyeria Informàtica. Universitat Autònoma de Barcelona.

[Salvià, 1997] Salvià, M. (1997). GeNoos: programació genètica en el llenguatge Noos. Master's thesis, Facultat d'Informàtica de Barcelona. Universitat Politècnica de Catalunya.

[Schreiber et al., 1993] Schreiber, G., Wielinga, B., and Breuker, J., editors (1993). *KADS: A Principled Approach to Knowledge-based System Development*, volume 11 of *Knowledge-based systems book series.* Academic Press.

[Serra, 1997] Serra, X. (1997). Musical sound modelling with sinusoids plus noise. In Poli, G. D., Picialli, A., Pope, S. T., and Roads, C., editors, *Musical Signal Processing.* Swets and Zeitlinger Publishers.

[Sierra et al., 1996] Sierra, C., Godo, L., López de Mántaras, R., and Manzano, M. (1996). Descriptive Dynamic Logic and its application to reflective architectures. *Future Generation Computer Systems*, 12:157–171.

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

[Slodzian, 1994b] Slodzian, A. (1994b). Knowledge level reflection. Master's thesis, Vrije Universiteit Brussel.

[Smith, 1985] Smith, B. C. (1985). Reflection and semantics in a procedural language. In Brachman, R. J. and Levesque, H. J., editors, *Readings in Knowledge Representation*, pages 31–40. Morgan Kaufmann, California.

[Smyth and Keane, 1995] Smyth, B. and Keane, M. T. (1995). Remenbering to forget: A competence-preserving case delection policy for case-based reasoning systems. In *Proceedings of IJCAI-95*, pages 377–382.

[Steele, 1990] Steele, G. L. (1990). *Common Lisp, the language. Second edition.* Digital Press.

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

[Studer et al., 1996] Studer, R., Eriksson, H., Gennari, J., Tu, S., Fensel, D., and Musen, M. (1996). Ontologies and the configuration of problem-solving methods. In *Tenth Knowledge Acquisition for Knowledge-Based Systems Workshop.*

[Tambe et al., 1990] Tambe, N., Newell, A., and Rosenbloom, P. (1990). The problem of expensive chuncks and its solution by restricting expressiveness. *Machine Learning*, 5:299–349.

[Taube, 1991] Taube, H. (1991). Common Music: a music composition language in Common Lisp and CLOS. *Computer Music Journal*, 15(2):21–32.

[Taube, 1996] Taube, H. (1996). Common Music. Technical report, School of Music, University of Illinois. online at <http://ccrma-www.stanford.edu/CCRMA/Software/cm/cm.html>.

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

[Van de Velde, 1994a] Van de Velde, W. (1994a). An overview of commonKADS. In Breuker, J. and Van de Velde, W., editors, *CommonKADS library for expertise modelling.* IOS Press.

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

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

[van Marcke, 1987] van Marcke, K. (1987). KRS: An object-oriented representation language. *Revue d'Intelligence Artificielle*, 1(4):43–68.

[van Marcke, 1988] van Marcke, K. (1988). *The use and implementation of the representation language KRS*. PhD thesis, Vrije Universiteit Brussel.

[Veloso, 1992] Veloso, M. (1992). *Learning by analogical reasoning in general problem solving*. PhD thesis, Carnegie Mellon University.

[Veloso and Carbonell, 1993] Veloso, M. and Carbonell, J. (1993). Toward scaling up machine learning: A case study with derivational analogy in PRODIGY. In Minton, S., editor, *Machine Learning Methods for Planning*, pages 233–272. Morgan Kaufmann.

[Weyhrauch, 1980] Weyhrauch, R. (1980). Prolegomena to a theory of mechanized formal reasoning. *Artificial Intelligence*, 13:133–170.

[Wielinga et al., 1992] Wielinga, B., Schreiber, G., and Breuker, J. (1992). KADS: A modelling approach to knowledge engineering. *Knowledge Acquisition*, 4(1):5–54. Special Issue 'The KADS approach to knowledge engineering'.

[Wielinga et al., 1993] Wielinga, B., Van de Velde, W., Schreiber, G., and Akkermans, H. (1993). Towards a unification of knowledge modelling approaches. In David, J. M., Krivine, J. P., and Simmons, R., editors, *Second Generation Expert Systems*, pages 299–335. Springer Verlag.

[Wray et al., 1995] Wray, R., Chong, R., Philips, J., Rogers, S., and Walsh, B. (1995). A survey of cognitive and agent architectures. Technical report, University of Michigan. <url:http://ai.eecs.umich.edu:80/cogarch0/>.