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/non-monotonic).

These design decisions then lead to the support of specific *capabilities* such as capabilities related to problem solving (e.g. planning, self-reflection, or meta-reasoning) 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 $(ML)^2$ [van Harmelen and Balder, 1992] were developed. Below we will describe the REFLECT approach by means of the description of the $(ML)^2$ 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.