Developing Knowledge-Based Systems with MIKE

J. Angele, D. Fensel, D. Landes[*], and R. Studer
Institute AIFB, University of Karlsruhe
76128 Karlsruhe, Germany,
tel. [049] (0)721 608 3923, fax [049] (0)721 693717
e-mail: {angele | fensel | studer}@aifb.uni-karlsruhe.de

Abstract

The paper describes the MIKE (*Model-based and Incremental Knowledge Engineering*) approach for developing knowledge-based systems. MIKE integrates semiformal and formal specification techniques together with prototyping into a coherent framework. All activities in the building process of a knowledge-based system are embedded in a cyclic process model. For the semiformal representation we use a hypermedia-based formalism which serves as a communication basis between expert and knowledge engineer during knowledge acquisition. The semiformal knowledge representation is also the basis for formalization, resulting in a formal and executable model specified in the Knowledge Acquisition and Representation Language (KARL). Since KARL is executable, the model of expertise can be developed and validated by prototyping. A smooth transition from a semiformal to a formal specification and further on to design is achieved because all the description techniques rely on the same conceptual model to describe the functional and non-functional aspects of the system. Thus, the system is thoroughly documented at different description levels, each of which focuses on a distinct aspect of the entire development effort. Traceability of requirements is supported by linking the different models to each other.

Keywords

Knowledge Engineering, Knowledge Acquisition, Knowledge-based Systems, Domain Modeling, Task Modeling, Problem-Solving Method

---

[*] Author's current address: Daimler-Benz Research and Technology, Dept. Software Engineering (F3K/S), D-89013 Ulm, tel. [049] (0)731 505 2869, e-mail: landes@dbag.ulm.DaimlerBenz.COM

1 INTRODUCTION

Edward A. Feigenbaum (1977) defines the activity of knowledge engineering as “the art of building complex computer programs that represent and reason with knowledge of the world.“ In the last twenty years, significant progress has been achieved in improving the understanding of the knowledge engineering activity and in providing methods and tools that shifted knowledge engineering from an art to an engineering activity. Basically, two waves can be identified, which we will call the knowledge *transfer* period and knowledge *modelling* period.

During the *knowledge transfer* period a number of tools were developed which should support the rapid transfer process of human knowledge into implemented systems. Rule-based interpreters which support the rapid implementation of knowledge-based systems (KBSs) were developed (cf. Hayes-Roth et al., 1983). However, the elicitation of knowledge was soon identified as the main bottleneck in developing such systems. A number of knowledge elicitation tools were developed to support the mining process for the knowledge nuggets. Two implicit assumptions were made by these approaches: (1) Production rules systems are the correct and adequate level for representing such knowledge and (2) the knowledge that is required by a knowledge-based system already exists and only needs to be collected.

The above techniques worked well in quickly deriving small prototypical systems; they failed however to produce large, reliable and maintainable KBSs. In reaction to these problems both assumptions were criticized. For example, Clancey (1983) discussed the limitations of production rules and Wielinga & Breuker (1984) proposed a modelling point of view on knowledge acquisition. That is, knowledge acquisition is not the elicitation and collection of already existing knowledge pieces, but the process of creating a knowledge model which did not exist beforehand. In consequence, knowledge engineering is no longer seen as a direct transfer process of human knowledge in a production rule implementation. Instead, a *model* of the required functionality of the KBS and the knowledge that is required to achieve the functionality is built to bridge the large gap between informal requirements and human expertise on the one hand and an implemented system on the other hand. Meanwhile, the modelling paradigm has become quite dominant in the knowledge engineering area and a number of principles are shared by most approaches.

- Separating a domain model from a task model. A task model describes the problem that should be solved by a KBS. A generic task (Chandrasekaran et al., 1992) abstracts from the specific application domain and is therefore applicable to (i.e., reusablein) a broad range of domains. A diagnostic task can be described independently from whether it is performed to diagnose cancer, heart diseases, or faults of mechanical devices. In the same sense, domain models can be used (i.e., reused) for a number of tasks. Building a domain model is expensive and knowledge sharing and reuse can significantly reduce development costs (cf. Farquhar et al., 1997; van Heijst et al., 1997). A functional component model of a technical artefact can be used for design tasks or to diagnose a malfunction using model-based diagnosis techniques (de Kleer et al., 1992).
- Modelling the problem-solving process is part of the knowledge model. In textbooks on software engineering, authors propose a clean distinction between what the system should achieve and how it achieves it. The former is the specification and the latter is realized during design and implementation. In KBS development a more differentiated point of view is taken. That is, not only the *what* , but also an implementation independent description of the *how* , is a necessary part of a specification of a KBS. In general, most problems tackled with knowledge-based systems are inherently complex and intractable (see Fensel & Straatman, 1996; Nebel, 1996). A specification of a KBS has to

describe not just a realization of the functionality, but one which takes into account the constraints of the reasoning process and the complexity of the task. *Problem-solving methods* (cf. Breuker & Van de Velde, 1994) link domain and task models by describing how the domain knowledge can be used to achieve the goals of the task in an efficient manner.

In this paper we present one of these approaches: the MIKE approach (Model-based and Incremental Knowledge Engineering), which provides a development method for KBSs covering all steps from the initial elicitation through specification to design and implementation. MIKE proposes the integration of *semiformal* and *formal specification techniques, prototyping,* and *life cycle models* into an engineering framework:

- *Informal and semiformal models* of the knowledge provide a high and informal level for knowledge description. Graphical means like entity-relationship diagrams, data flow diagrams, flow charts, and state-transition diagrams, are used. This type of information is easy to understand and very useful as a mediating representation for the communication between the domain expert, the user, and the system developer. The semiformal models are represented in a hypermedia-based formalism (Neubert, 1993).
- The language KARL (Knowledge Acquisition and Representation Language, (Fensel et al., to appear (a)) allows the functionality as well as the knowledge necessary to achieve the functionality to be described in an unambiguous and precise manner. The language DesignKARL (Landes, 1994) provides additional primitives that allow the implementation-oriented aspects of the system to be modelled.
- Because KARL is an executable language the specification may be developed by a prototyping approach, i.e. it may be evaluated by a running prototype. Often, this is nearly the only way to arrive at realistic descriptions of the desired functionality as well as a judgement of the competence of the captured knowledge.
- Additional representations the modelling decisions made during the various phases of the life-cycle (Landes & Studer, 1995) to be documented. This enables requirements to be traced back, i.e. it is recorded which parts of the implementation are addressing a particular requirement.

All different activities of the building process itself are embedded in a cyclic life cycle model reflecting the incremental, cyclic and highly reversible nature of knowledge acquisition (Morik, 1987). Integrating prototyping and support for an incremental and reversible system development process into a model-based framework is actually the main distinction between MIKE and other model-based approaches like CommonKADS (Schreiber et al., 1994):

- MIKE took the *model of expertise* of CommonKADS as its general model pattern and provides a smooth transition from a semiformal model to a formal model and further to the design model of the system. Each of these different documents represent the model of expertise at different levels of detail. The smooth transition between the different model types is an essential support enabling incremental and reversible system development in practice.
- The executability of KARL enables validation of the models by prototyping. From our point of view, prototyping is an essential means to clarify weak requirements and to establish a notion of the competence of the modelled knowledge.

Though the MIKE approach addresses the building process of KBSs, these principles and methods apply to requirements analysis and the specification of information systems and “conventional“ software systems as well. For instance Fensel et al. (1993) show how KARL may be used to formalize the system model of Structured Analysis and Angele (1996) shows how KARL may be used for conceptual modelling of information systems.

The paper is organized as follows. Section two introduces the overall process model of

MIKE. Section three discusses the documents and models which are the results of the different MIKE development steps. Section four sketches the tool support that is provided by MIKE and section five discusses related work. Section 6 draws some conclusions.

2 THE DEVELOPMENT PROCESS

In this section we will describe the MIKE knowledge engineering *process*. First, we will sketch the underlying modelling paradigm. Then we will describe the different activities and their resulting documents and the sequence in which these activities are performed. Finally, we will discuss the use of prototyping and reuse as two guiding principles of such a development process.

2.1 The Model of Expertise

The distinction of symbol level and knowledge level (Newell, 1982) gives rise to a separation of the description of a KBS at the knowledge level from the description at the symbol level. A similar separation is made in software engineering where a specification of a system should be clearly separated from its design and implementation. In an extension of such a specification, a knowledge level model does not only describe the functionality of the system but also describes at a high level of abstraction *how* the system provides this functionality. This is done for the following reasons:

- As already mentioned, most problems tackled with KBSs are intractable in their general definition (Nebel, 1996; Fensel & Straatman, 1996). Therefore, it is necessary to model problem restriction as well as heuristics that improve the efficiency of the system. Pragmatic restrictions as well as “clever heuristics“ are often exactly the knowledge that distinguishes an expert from a novice (VanLehn, 1989).
- Typically, the necessary functionality of a KBS is only available as procedural knowledge of how to perform the task. Therefore, specifying the functionality of a KBS is only possible as result of modelling how it is achieved.

In consequence, it is not sufficient to give a detailed functional specification of a KBS and to build a solution using ’normal’ computer science know-how. Instead task and domain specific knowledge and problem-solving methods that make use of this knowledge are necessary to build a solution.

According to the KADS approach the knowledge level description of a KBS is called the *Model of Expertise* (MoE) (cf. Schreiber et al., 1994). This model separates different kinds of knowledge at different layers:

- The *domain layer* contains knowledge about domain-specific concepts, their attributes and their relationships and it contains domain specific heuristics for the problem-solving process.
- The *inference layer* contains the knowledge about the functional behavior of the problem-solving process. This layer indicates which inferences are necessary and which data dependencies exist between them.
- The *task layer* (called *control* layer in MIKE) contains knowledge about the goals of a task and the control knowledge required to perform the task. This layer specifies the sequence of the inferences within the problem-solving process.

The problem-solving process is represented at the inference and control layer. Thus these two layers together with the heuristics at the domain layer describe how the system pro-

vides its functionality.

This structure and the building primitives of the MoE are used throughout the entire development process in MIKE. So all development activities work on the same conceptual model and all documents resulting from these activities are based on this same conceptual model. During the succeeding realization process the structure of this model should be preserved as best as possible. This allows parts of the final expert system to be traced back to corresponding descriptions of earlier development steps.

2.2 Steps and documents

The different development activities and the documents resulting from these activities are shown in Figure 1. Within the entire development process a large gap has to be bridged between the informal requirements and human knowledge on the one hand and the final realization of the expert system on the other hand. Dividing this gap into smaller ones reduces the complexity of the entire modelling process because in every step different aspects may be considered independently from other aspects. Therefore in MIKE the entire development process is divided into a number of subactivities: *Elicitation*, *Interpretation*, *Formalization/ Operationalization*, *Design,* and *Implementation*. Each of these activities deals with different aspects of the entire system development.

The knowledge acquisition process starts with *Elicitation* (see Figure 1), i.e. trying to get hold of the experts' knowledge. Methods like structured interviews, observation, structuring techniques etc. (Eriksson, 1992) are used for acquiring informal descriptions of the knowledge about the specific domain and the problem-solving process itself. The resulting knowledge expressed in natural language is stored in so-called knowledge protocols.

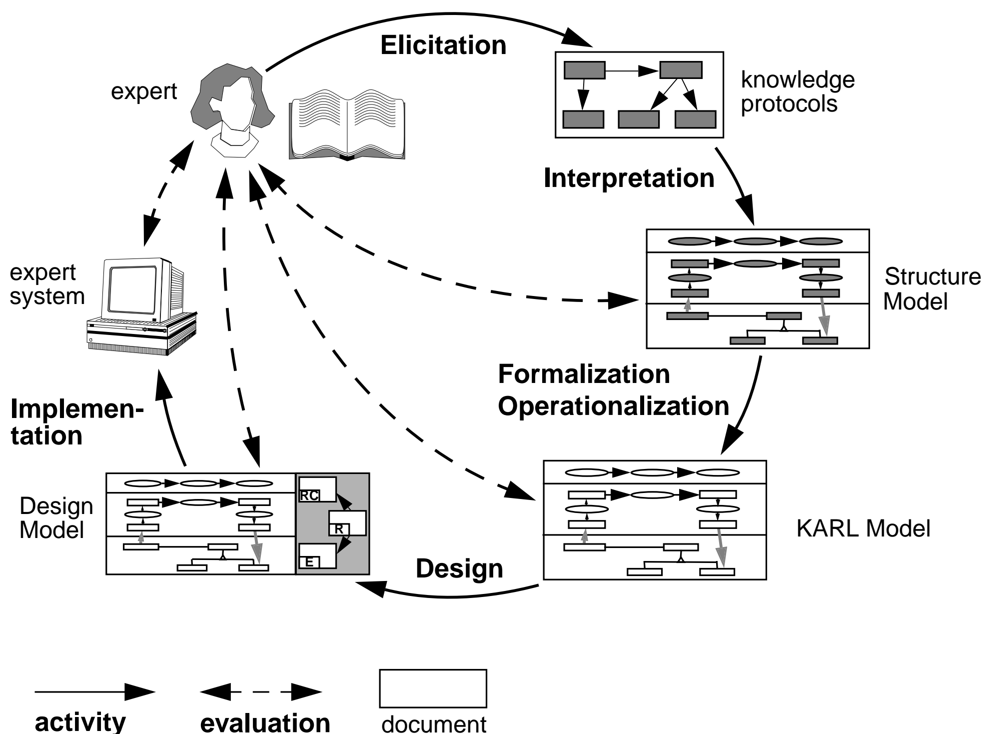

Figure 1 Steps and documents in the MIKE development process

During the *Interpretation* phase the knowledge structures which may be identified in the knowledge protocols are represented in a semi-formal variant of the MoE: the *Structure Model* (Neubert, 1993). All structuring information in this model, like the data dependencies between two inferences, is expressed in a fixed, restricted language while the basic building blocks, e.g. the description of an inference, are represented by unrestricted texts. This representation provides an intitial structured description of the emerging knowledge structures. It has a mediating role between the natural language descriptions and the formal MoE. The development of mediating representations provides various advantages: Semi-formal representations can be used as a communication level between the knowledge engineer and the expert. The expert can be integrated in the process of structuring the complex knowledge such that the knowledge engineer is able to interpret and formalize it more easily. Thus, the cooperation between expert and knowledge engineer is improved and the formalization process is simplified. An early evaluation process is possible in which the expert himself is integrated. In addition, a mediating representation is a basis for documentation and explanation. The maintenance of the system is also simplified.The Structure Model may be evaluated by semi-automatical simulation.

This semi-formal MoE is the foundation for the *Formalization/Operationalization* process which results in the formal MoE: the KARL Model. This formal MoE has the same structure as the semi-formal MoE while the basic building blocks which have been represented as natural language texts are now expressed in the formal language *KARL* (Fensel et al., to appear (a)). This representation avoids the vagueness and ambiguity of natural language descriptions. Because the former texts have to be interpreted, the formalization helps to get a clearer understanding of the entire problem-solving process. This formalized MoE can be directly mapped to an operational representation because KARL (with some small limitations) is an executable language. Thus the KARL Model can be evaluated by testing.

The result of the knowledge acquisition phase, the KARL Model, captures all functional requirements for the final expert system. During the *Design* phase additional non-functional requirements are considered. These non-functional requirements include e.g. efficiency, maintainability, robustness, portability, etc., but also the constraints imposed by target software and hardware environments. Efficiency is already partially covered in the knowledge acquisition phase, but only to the extent as it determines the problem-solving method. Consequently, functional decomposition is already part of the knowledge acquisition phase. Therefore, the design phase in MIKE constitutes the equivalent of detailed design and unit design in software engineering approaches. The *Design Model* which is the result of this phase is expressed with the language *DesignKARL* (Landes, 1994). DesignKARL extends KARL by providing additional primitives for structuring the model and for describing algorithms and data types. Similar to the MoE, the Design Model can be evaluated through testing. DesignKARL additionally allows the design process itself and interactions between design decisions to be described.

The Design Model captures all functional and non-functional requirements posed to the KBS. In the *Implementation* process the Design Model is implemented in the target hardware and software environment.

The result of all phases is a set of several interrelated refinement states of the MoE. The knowledge in the Structure Model is related to the corresponding knowledge in the knowledge protocols via *elicitation links*. Concepts as well as inference actions are related to protocol nodes, in which they have been described using natural language. The Design Model refines the KARL Model by refining inferences into algorithms and by introducing additional data structures. These parts of the Design Model are linked to the corresponding inferences of the MoE and are thus in turn linked to the knowledge protocols (Landes, 1994). Combined with the goal of preserving the structure of the MoE during design, the links be-

tween the different model variants and the final implementation ensure traceability of (non-)functional requirements.

2.3 Prototyping

The development process of MIKE inherently integrates different types of prototyping (Angele et al., 1996a). The entire development process, i.e. the sequence of knowledge acquisition, design, and implementation, is performed in a cycle guided by a *spiral model* (Boehm, 1988) as process model. Every cycle produces a prototype of the expert system which may be evaluated by testing it in the real target environment. The results of the evaluation are used in the next cycle to correct, modify, or extend this prototype. This process is continued until all requirements posed to the KBS are fulfilled. In contrast to a linear process model this allows even changing requirements to be taken into account. This cyclic approach also integrates the maintenance of the system into the development process. Maintenance is performed by evaluating the current system and subsequently performing a set of development cycles in order to correct the system or to adapt it to changed requirements.

The executability of the language KARL allows the MoE to be built by explorative prototyping. Thus the different steps of knowledge acquisition are performed cyclically until the desired functionality has been reached. Thus the result of knowledge acquisition has been evaluated by the expert and therefore provides a well-founded basis for the succeeding realization process.

In a similar way, non-functional requirements are tackled by experimental prototyping in the design phase. DesignKARL can be mapped to an executable version which allows the Design Model to be evaluated by testing. The Design Model is refined by iterating the subphases of the design phase until all requirements are met.

Because of the possibility to clearly separate the modeling of the functional requirements in knowledge acquisition, the non-functional requirements in the design phase, and the implementation in the implementation phase, the development activities may focus separately on different aspects: this divide-and-conquer strategy considerably reduces the complexity of the entire development process.

2.4 Reuse

The above described development process may be supplemented by reusing parts of models developed earlier for similar problems.

The method for tackling the given task is represented at the control layer and the inference layer of the MoE. These two parts together are called problem-solving method (PSM)(Breuker & Van de Velde, 1994; Puppe, 1993). A PSM decomposes the given task into a hierarchy of subtasks. A method is assigned to each subtask which solves this subtask. Each method in turn may pose new subtasks until a level of elementary methods is reached. A PSM generically describes the different roles which are played by different kinds of domain knowledge during the problem-solving process. This may be exploited by using a PSM as a guideline to acquire static domain knowledge and additional domain-specific problem-solving knowledge like heuristics and constraints. A PSM may be represented independently of the domain it is applied in. For instance the PSM Hierarchical Classification which classifies an object in a hierarchy of object classes may be applied to classify cars or to classify animals. So for another problem which may be solved by Hierarchical Classification the PSM may be reused. Reusing and adapting PSMs is based on the assumptions they introduce on the required domain knowledge and the restriction of the task (Benjamins et al., 1996). How to organize a

library of PSMs and how to support the stepwise refinement process of PSMs is described in (Fensel, 1997b).

In a similar way, the construction of the domain layer of the MoE is supported by ontologies. An *ontology* provides an explicit specification of a conceptualization (Gruber, 1993). In essence, two types of ontologies may be distinguished:

- *Commonsense ontologies* aim at defining a collection of top-level concepts and associated relations which are independent of a specific application domain. In order to make the development of these ontologies manageable, they are typically divided into different clusters, e.g. for time and events, space or quantities (compare e.g. Lenat & Guha, 1990; Knight & Luk, 1994).
- *Domain ontologies* provide a conceptualization of a specific domain. They constrain the structure and contents of particular domain knowledge and define the vocabulary which is used to model the domain (van Heijst et al., 1997). Domain ontologies aim at making domain knowledge shareable across different tasks and reusable for different tasks. Farquhar et al. (1997) describe the Ontolingua server which supports composing complex ontologies from a library of already available ontologies.

Appropriate methods are required for supporting the reuse of ontologies. Pirlein & Studer (1994) introduce the KARO approach, which offers an integrated set of formal, lexical, and graphical methods for reusing ontologies and adapting them to the new task context. In general, a reuse-oriented approach should support both the reuse of PSMs and the reuse of ontologies. Pirlein & Studer (1997) discuss, how the reuse of PSMs may be integrated with the reuse of ontologies to support the construction of the MoE.

3 THE DOCUMENTS AND THEIR REPRESENTATIONS

In this section we present in detail the documents and models which are the results of the different MIKE development steps. These models describe the various types of knowledge at different levels of formality.

3.1 Knowledge Protocols

The MIKE development process begins with the Elicitation step. In this step various types of interviews are conducted. In addition, relevant sections of textbooks as well as Web documents and other kinds of documents are collected. The results of all these activities are stored in so-called *knowledge protocols* which contain natural language descriptions of the acquired knowledge (Neubert, 1993). In addition, administrative information like the date or the participants of the interview is stored. Knowledge protocols may be connected to each other by using two types of links: an *ordering link* specifies the temporal order in which two protocols have been collected, whereas a *refinement link* indicates that one protocol is a refinement of another one, i.e. is describing a specific subject in more detail.

Subsequently, we will use a running example for illustrating the MIKE documents and models: the task is to select appropriate methods for dismantling a building which allow at least parts of the building rubble (Fichtner et al., 1995) to be recycled. The task was tackled in cooperation with an external partner[*] in order to evaluate the MIKE approach in a complex domain. In Figure 2 we show parts of three knowledge protocols which resulted from

[*] Institute for Industrial Production, University of Karlsruhe

various interviews with the domain experts.

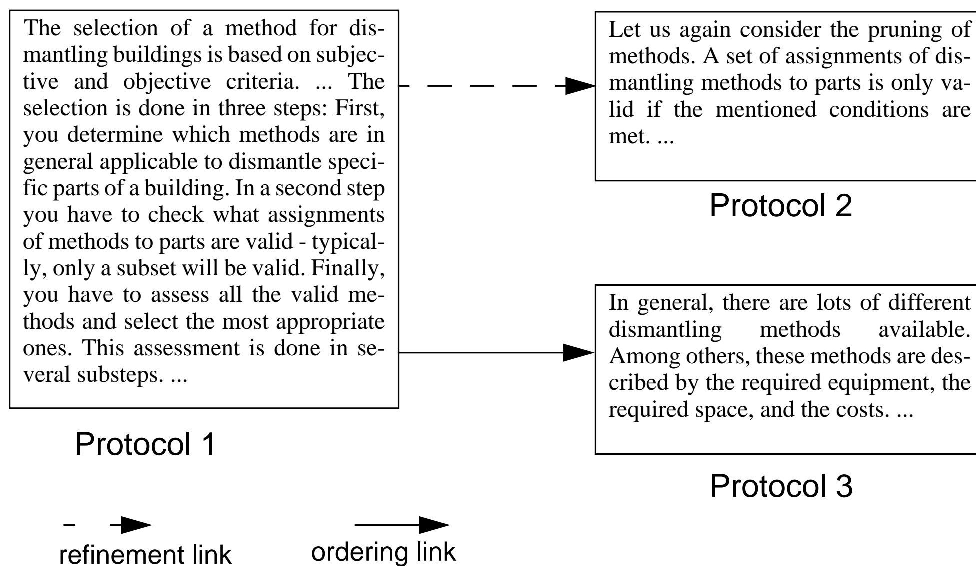

Figure 2 Knowledge Protocols (translated in English for the purpose of the paper)

Knowledge protocol 2 is connected by a refinement link to knowledge protocol 1 since it describes the pruning of methods in more detail. Furthermore, we have an ordering link between knowledge protocol 1 and knowledge protocol 3, because protocol 3 is based on an interview which had been conducted later in the elicitation process.

3.2 The Common Conceptual Model: The Model of Expertise

Throughout the MIKE development process a common conceptual model is used for describing the functionality of the KBS under development: the *Model of Expertise* (MoE). The MIKE Model of Expertise is derived from the *KADS model of expertise* (Schreiber at al., 1993). It is divided into different layers, each containing a different type of knowledge. The MIKE Model of Expertise distinguishes three types of knowledge at three different layers. These types establish a static view, a functional view, and a dynamic view to the KBS.

Domain knowledge at the *domain layer* consists of static knowledge about the application domain. The domain knowledge should define a conceptualization of the domain as well as task-specific heuristics.

Inference knowledge at the *inference layer* specifies the *inferences* that can be made using the domain knowledge and the *knowledge roles* which model input and output of the inferences. Three types of knowledge roles are distinguished. Roles which supply domain knowledge to an inference action are called *views*, roles which model the data flow dependencies between inference actions are called *stores*, and roles which are used to write final results back to the domain layer are called *terminators*. The inferences and roles together with their data flow dependencies constitute a description of the problem-solving method applied. The roles and the inference actions are specified in a *problem-solving-method-specific terminology* independently of the domain-specific terminology (cf. Studer et al., 1996).

A *domain view* specifies the relationship between the knowledge used at the inference layer and the domain-specific knowledge. It performs a transformation of the domain-specific

terminology of the domain layer into the terminology specific to the problem-solving method at the inference layer (cf. Gennari et al., 1994).

Dynamic control knowledge at the *control layer* is used to specify *control* over the execution of the inferences of the inference layer. It determines the sequence in which the inferences are activated.

Both inference and control knowledge are domain independent, i.e. they describe the problem-solving process in a generic way using a terminology specific to the problem-solving method.

3.3 The Semi-formal Model of Expertise: The Structure Model

In MIKE the knowledge acquisition phase results in a formal specification of the Model of Expertise using the formal specification language KARL. However, the construction of this formal specification is divided into two steps: first, a semi-formal description of the Model of Expertise is constructed in the Interpretation phase, resulting in the so-called Structure Model. Using the Structure Model as a starting point, the formal KARL Model is constructed in the Formalization/Operationalization step.

The *Structure Model* is described in a hypertext-based formalism offering various types of *nodes* and *links*. Each node is inscribed by natural language text (Neubert, 1993).

On the domain layer, nodes represent domain *concepts* and links represent various types of relationships between concepts: *generalization*, *aggregation*, and *domain specific relationships*. The graphical notation for the describing the domain layer has been adopted from OMT (Rumbaugh et al., 1991).Thus, concepts are modeled as classes and relationships are modeled as associations.

Figure 3 shows both the Structure Model and the KARL Model of our running example (Fichtner et al., 1995). The domain terminology and domain knowledge needed to choose appropriate methods and techniques for dismantling a building are defined at the domain layer. For instance the available *dismantling methods* may among others be divided into *knocking down* and *blasting*. The *building parts* are represented in a part-of hierarchy. It should be clear that the domain layer contains additional knowledge, e.g. the conditions which enable or disable certain techniques for the case at hand.

It is important to note that the inscriptions of the nodes of the Structure Model are directly copied from the knowledge protocols. For instance the inscription of the node *dismantling method* is taken from knowledge protocol 3. As a consequence, an elicitation link is established between this node and protocol 3.

On the inference layer nodes represent the inference steps and knowledge roles. In our example the global inference action *Select Dismantling Method* is decomposed into three top-level inference actions (compare knowledge protocol 1 in Figure 2). Therefore, we find three inference actions on the inference layer: the inference action *Assign* determines all methods which are applicable for demolishing or removing the different parts of the building and assigns them to the appropriate parts. Therefore, *methods* and *parts* are used as input roles for *Assign*. *Prune* eliminates part-methods pairs which are disabled by case specific conditions. Here again, we see that the description of the inference action *Prune* is copied from one of the knowledge protocols. *Assess* then assesses the remaining part-method pairs and chooses those methods which are best suited and produce the least costs. *Assess* is a complex inference action, i.e. it is hierarchically refined until elementary inference steps are reached.

The control flow between these inferences is defined at the control layer. In general, *se-*

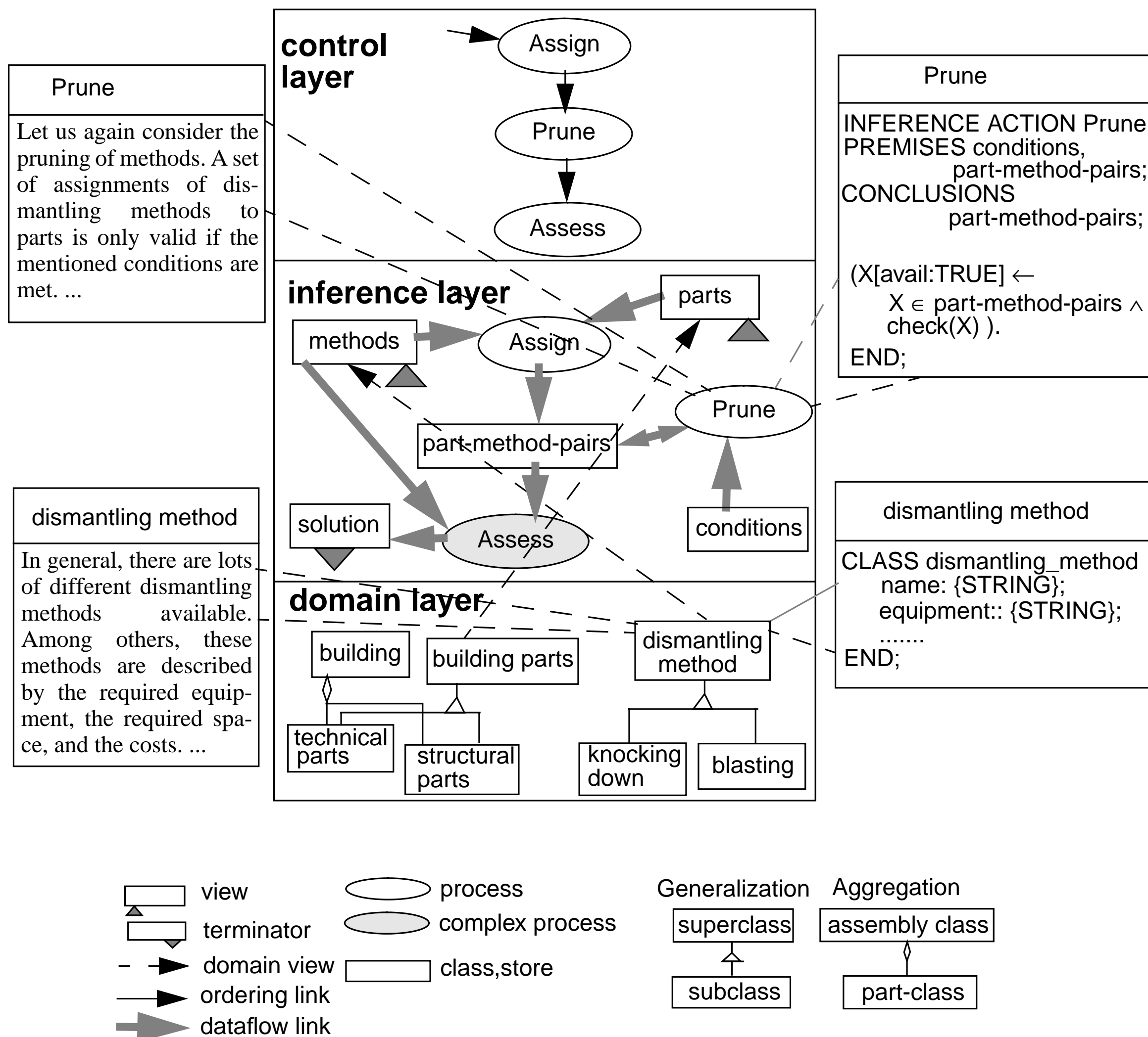

Figure 3 Model of Expertise for the task of selecting methods to dismantle a building. At the left the semi-formal variant (Structure Model), at the right the formal variant (KARL Model) is shown.

quence, *loop*, and *alternative* are used to specify the control flow. In our example the control flow is very simple: the three top-level inference actions are executed once in the order *Assign*, *Prune*, and *Assess*.

As an extension to the KARL Model, the Structure Model includes an additional layer for modeling non-functional requirements (NFRs): the so-called *NFR layer*. This is due to the fact that the KARL Model provides a formal specification of the *functional* behavior of the KBS, whereas non-functional requirements are used during *Design* to derive the Design Model from the KARL Model.

The *NFR layer* is used for describing the NFRs the system must fulfill. Basically, node types like *requirements category* or *evaluation criterion* as well as link types like *correlation* or *conflict solution* are used to semi-formally describe types of requirements and various relationships which may exist between them (see Landes & Studer, 1995 for details). In Figure 4 a partial NFR context for our running example is shown. We assume that we find in a knowledge protocol the statement that the resulting KBS would solve the problem faster than

a human expert. This statement is interpreted as a runtime requirement. Thus, an instance of the requirement category *runtime efficiency* is created on the NFR layer and linked to the corresponding text fragment in the knowledge protocol. Since the text fragment also contains a criterion to check whether the requirement has been met (the KBS should need less than 2 hours), a node of type *evaluation criterion* is created as well. In addition, we find a *reference link* to the global inference action *Select Dismantling Method* since the requirement affects all the steps which have to be carried out for solving our problem.

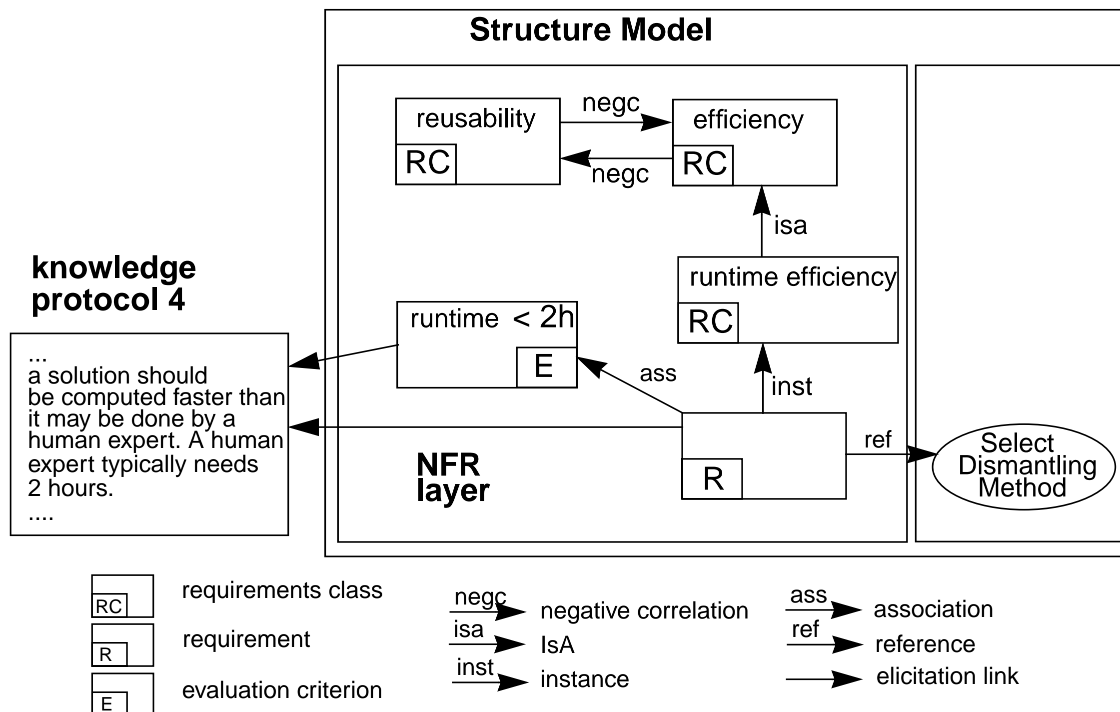

Figure 4 Non-Functional Requirements for the task of selecting methods for dismantling a building

The graphical representations of the control flow, the data flow, and the static knowledge have been adopted from corresponding representations in software engineering (program flow diagrams, data flow diagrams, OMT notation), and the representation of non-functional requirements is similar to corresponding graphical representations in software engineering and information systems engineering. The Structure Model is an adequate basis for the development of the KARL Model as well as of the Design Model: it comprises the knowledge about the functional aspects as well as non-functional aspects, the domain knowledge is clearly separated from the problem-solving knowledge, and all the different parts of the Structure Model are well-integrated and linked to the appropriate parts of the knowledge protocols.

3.4 The formal Model of Expertise: the KARL Model

In this section we describe the formal Model of Expertise, the KARL Model, represented in the language KARL. A detailed description of KARL may be found in (Fensel et al., to appear (a)). The main characteristics of KARL are the combination of a *conceptual description* of a KBS with a *formal* and *executable specification*. In the following we give a brief over-

view of KARL and its use to specify the different types of knowledge at the different layers of the KARL Model.

Logical-KARL (L-KARL)

L-KARL enriches the modelling primitives of first-order logic through the inclusion of epistemological primitives but preserves its model-theoretical semantics. These additional primitives allow static aspects to be described more adequately than pure first-order logic. In this way, ideas of semantical and object-oriented data models are integrated into a logical framework enabling the declarative description of terminological as well as assertional knowledge.

L-KARL distinguishes classes, objects, and values. Classes which are arranged in an is-a hierarchy with multiple attribute inheritance are used to describe terminological knowledge at the domain layer and in knowledge roles at the inference layer. In Figure 1 dismantling methods are described by a class hierarchy:

```
CLASS dismantling_method            CLASS knocking_down
      name: {STRING};                     ISA dismantling_method;
      equipment::{STRING};                space: {INTEGER};
      ....                                ....
END;                                END;
```

The attributes have range restrictions described within the curved brackets: for the attribute *name,* for instance, only strings are allowed as values. *Equipment* is a set-valued attribute, i.e. it has a set of values assigned. The class *knocking_down* inherits all attributes from its superclass *dismantling_method*.

Intentional and factual knowledge is described by logical formulae dealing with classes, instances of classes, and values. Such formulae are used to describe (i) facts and sufficient conditions at the domain layer, (ii) to declaratively describe the input-output behavior of inference actions at the inference layer, and (iii) to represent views and terminators, i.e. to transform domain specific knowledge into generic knowledge needed by the problem-solving method and vice versa.

The basic ingredients of logical formulae in KARL are F-atoms:

- $e \in c$ is an *element-atom* where e is an element identifier and c is a class identifier. An element-atom states that an object e is an element of class c.
- $c \leq d$ is an *is-a-atom*, where c and d are class identifiers. An is-a-atom expresses the fact that a class c is a subclass of class d. Using variables within c or d it is possible to browse through the class hierarchy using this term.
- $o[..., a{:}T, ..., s{::}\{S_1,..,S_n\}, ...]$ is a data-atom, where o is either an element identifier or a class identifier, T, S_i are data-atoms. a is an attribute name of a single-valued attribute, s of a set-valued attribute. A data-atom defines attribute values for the object which is referred to by the identifier o.
- $e = d$ denotes an *equality*-atom, where e and d are identifiers. This means that e and d denote the same element, class, or value.

In addition, *P-atoms* $p(a_1{:}T_1, ..., a_n{:}T_n)$ allow the expression of relationships between objects T_i in a similar way as in predicate logic. The arguments of a predicate are named by the a_i´s in order to improve readability. A set of built-in P-atoms is available for instance for mathematical computations.

Instances of classes are described by facts (F-atoms without variables). Objects are represented by unique identifiers. For instance a knocking down method with identifier *km1* may be given by the following fact (data-atom):

km1[name: knock1, equipment:: {sphere}, space: 2,45]. // *method with attributes*

To express the fact that km_1 is an element of the class *knocking_down* the following fact (element-atom) is used:

$km_1 \in$ *knocking_down.* // km_1 *is an instance of knocking_down*

Facts are used for representing factual knowledge at the domain layer. In addition the contents of stores at the inference layer consist of sets of facts.

Logical formulae are built from F-atoms in the usual way using logical connectors $\wedge$ (and), $\vee$ (or), $\neg$ (not), $\leftarrow$ (implication), brackets, and variable quantification.
For instance the inference action *Prune* in Figure 1 is represented by the rule:

$X[avail{:}TRUE] \leftarrow X \in part\text{-}method\text{-}pairs \wedge check(X).$

This rule checks all *part-method pairs* to determine whether they are suitable via the predicate *check* (which is defined at the domain layer) and marks them as available (*avail*: *TRUE*).

The following rule determines all subclasses of the class *dismantling_method* by using an is-a-atom in its rule body:

$sub(c{:}\ X) \leftarrow X \leq dismantling_method.$

One resulting instance of the P-atom *sub* in our example is *sub*(*c*:*knocking_down*).

The logical language KARL is restricted to Horn logic with equality extended by stratified negation. Using a logical language enriched by additional primitives allows the description of the respective parts of the model on an adequate abstraction level and the restriction to Horn logic allows the KARL Model to be executed and thus supports the construction process by means of prototyping.

Procedural-KARL (P-KARL)

In KARL, knowledge about the control flow is explicitly described by the language P-KARL. The control flow at the control layer, i.e. the sequence of the activation of inference actions, is specified by the modelling primitives *sequence*, *loop*, and *alternative* which are similar to the control flow statements of procedural programming languages. The conditions for controling loops and alternatives consist of boolean expressions including boolean variables and boolean functions of the form *empty*(*s*,*c*). The boolean function *empty*(*s*,*c*) delivers the value true if the class *c* in store *s* at the inference layer contains no elements. For instance the simple control flow of our example in Figure 1 consists of a sequence of invocations of inference actions at the inference layer:

Assign;
Prune;
Assess;

The KARL Model

The two sublanguages of KARL, L-KARL and P-KARL, are used to represent the following parts of the KARL Model (see Figure 5).
L- KARL is used to represent:

- a hierarchy of classes, their relationships, and their attributes at the domain layer,
- sufficient and necessary conditions using logical formulae at the domain layer,
- the connections between the inference layer and the domain layer, i.e. views and terminators, using logical formulae (rules),
- the input-/output behavior of inference actions at the inference layer using logical formulae (rules),
- the internal structure of stores, views, and terminators at the inference layer by means of

classes, relationships between classes, and attributes of classes.
- facts at the domain layer and facts as conctents of stores at the inference layer.

P-KARL is used to represent:
- the control flow at the control layer.

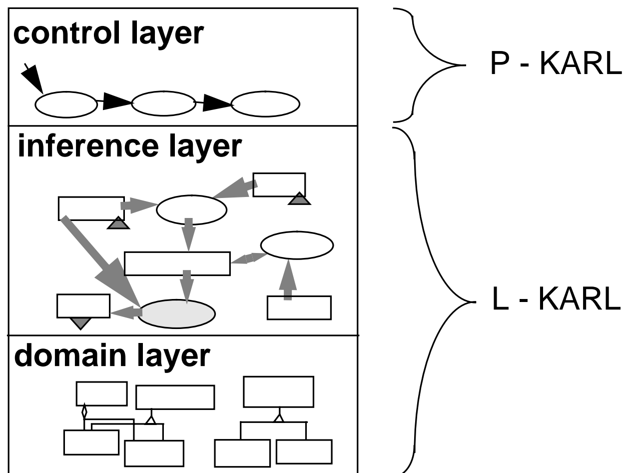

Figure 5 Representation of different parts of the KARL Model using the sublanguages of KARL.

KARL as a Formal and Executable Specification Language

The KARL Model contains the description of domain knowledge and knowledge about the problem-solving method (inference and control layer). The gist of the *formal semantics* of KARL is therefore the integration of static and procedural knowledge. For this purpose, two different types of logic have been used and integrated. The sublanguage L-KARL, which is based on object-oriented logics, combines frames and logic to define terminological as well as assertional knowledge. The sublanguage P-KARL, which is a variant of dynamic logic, is used to express knowledge about the control flow of a problem-solving method in a procedural manner. Both types of languages have been combined to specify the semantics of KARL. This semantics of KARL is defined by enriching the possible worlds semantics of Kripke structures (cf. Harel, 1984) with minimal model semantics of logic programs (cf. Ulman, 1988; Przymusinski, 1988) for defining possible worlds and elementary transition between worlds, see (Fensel, 1995; Fensel et al., to appear (a)) for more details. Based on this semantics, a constructive semantics and an optimized evaluation strategy have been developed which establish the foundation of an interpreter and debugger for KARL (Angele, 1993), (Fensel et al., to appear (a)). In this way it becomes possible to validate a formal specification by executing it and to create this specification using a prototyping approach.

3.5 The extended Model of Expertise: the Design Model

In this section, the Design Model and the underlying description language DesignKARL will be outlined. More details can be found in (Landes, 1994) and (Landes & Studer, 1995).

The Contents of the Design Model

The Design Model prescribes how the system has to be realized in order to comply to the functional and non-functional requirements posed against it. Thus, the Design Model encompasses the knowledge that is already contained in the MoE but adds information that forms the basis for the ensuing implementation of the system. In its final stage, the Design Model has to indicate precisely

- what the interfaces between the reasoning component and other components of the system look like,
- the constituents which make upf the reasoning component,
- what the interfaces and interactions between these constituents are, and
- how these constituents work internally, i.e. which algorithms and data structures have to be employed in which specific locations.

Basically, the design process can be viewed as a refinement of the information contained in the KARL Model, triggered by the non-functional requirements collected in the NFR layer of the Structure Model. Design decisions are constrained by the aim to preserve the structure of the KARL Model as far as appropriate. Structure-preserving design (Schreiber, 1993) is believed to facilitate a number of processes, e.g., maintenance, since portions of the design and implementation can be traced back to the KARL Model (or earlier models) more easily. Furthermore, structure-preserving design may support the system´s explanation capabilities since explanations may also resort to more intuitive and less implementation-dependent representations.

Maintainability is usually one of the major problems when building complex software systems. As a consequence, the Design Model does not pay attention exclusively to the artefact that will be the starting point for the implementation of a KBS, but also takes the process into account by which this artefact is constructed, in order to make sure that the rationale of design activities will not get lost. Therefore, the Design Model in MIKE consists of two parts, namely the *product model* and the *process model*.

Especially, the history of the design process has to be captured in addition to the motivations behind design decisions. The former is addressed by collecting and interrelating all the various versions of constituents in the product model that evolve during the design process. The latter aspect is covered in the process model. Since non-functional requirements motivate most of the design decisions in MIKE, they are an important aspect when trying to explicate the motivations for design decisions. A second major issue in regard to the process model is the description of design activities. Design activities are, on the one hand, the decomposition of goals, i.e. non-functional requirements, into subgoals and, on the other hand, elementary design decisions which directly affect specific portions of the product model and which fulfill subgoals that cannot be decomposed further.

Both parts of the Design Model are expressed using the same formalism, DesignKARL, which will be discussed in more detail in the following section.

DesignKARL: Language primitives for the product model

Since the MoE can be viewed as the initial stage of the product model, DesignKARL is a superset of KARL. Extensions introduced in DesignKARL for the description of the product model are targeted towards design-specific considerations, namely the definition of suitable data representations and algorithms and the definition of an appropriate system structure.

Data Types

An object class in KARL characterizes its elements by certain attributes, but abstracts from information which is irrelevant from a conceptual point of view. A key issue in the attempt

to improve efficiency is the introduction of data structures which exploit such additional information, e.g. an ordering of elements in a class or of values for a set-valued attribute, or support more efficient data access. Ideally, these data structures should also facilitate the mapping to the implementation environment. To these ends, DesignKARL supplies a collection of data types which currently comprises the types *value*, *class*, *predicate*, and *set* (which are already known in KARL) as well as the types *sequence*, *stack*, *queue*, *n-ary tree*, *hash table*, *index structure*, and *reference*. Instances of these data types can be introduced both at the domain layer and in roles at the inference layer.

For instance, in our running example of selecting appropriate dismantling methods, one factor that influences the appropriateness of a method is the cost it incurs. In order to retrieve the cheapest methods more efficiently, an instance of the data type sequence can be used in the store part-method-pairs (see Figure 5) such that it will include the possible associations of methods to parts of the building ordered by cost:

```
STORE part-method-pairs
        SEQUENCE part-methods-pairs OF part-method-assoc;
        // the instances of the class part-method-assoc are arranged as sequence
        CLASS part-method-assoc;
END;
```

Each data type is associated with pre-defined operations that may be performed on instances of the respective type for testing, data retrieval, and data storage. For instance, the data type sequence can be manipulated using operations which

- retrieve the first element of the sequence (*head*)
- return the sequence without its first element (*tail*)
- retrieve the data element located at a particular position of the sequence (*[]*)
- insert a data element into a particular position of the sequence (*insert*)
- remove a data element from a particular position of the sequence (*remove*)
- append a sequence to another one (*append*), or
- test whether a particular data item appears as an element of a sequence ($\in$, *index*).

DesignKARL only provides a restricted set of data types since a balance between simplicity and expressiveness has to be kept: on the one hand, the data types should be powerful enough to express the designer's intentions, on the other hand, a small collection of data types facilitates the largely automatic generation of appropriate code fragments which implement the respective data types for particular target environments. Data types available in DesignKARL are similar to those in common programming languages like C++ or Modula plus data types, such as hash tables and index structures, which are widely used in database applications. Therefore, the collection of data appears to be sufficient for most applications.

Besides supplying additional data types in DesignKARL, data types which are already known in KARL are extended with additional features. Classes, for example, can be associated with methods, i.e. user-defined operations. Methods may apply to the class as such (class methods) or to its elements (member methods) and can be specified either by means of logical expressions or by algorithms (see below). Subclasses and instances of a class inherit the methods defined in that class. Methods are used, e.g., for computing values of derived attributes of an object, but can also be employed to connect classes at the inference layer with data items at the domain layer.

DesignKARL: Algorithms

Besides employing appropriate data types, the design phase is concerned with developing suitable algorithms. While knowledge acquisition focuses on specifying which processing steps have to be carried out in principle to solve the given task, this description may need to

be refined and, possibly, restructured in order to be realized efficiently. Thus, an algorithm in DesignKARL need not necessarily be a sophisticated algorithm from a textbook, but may simply result from adding control information (which had previously been neglected since it was not required from a conceptual point of view) to the description of an inference action. The description of algorithms may be carried out similarly to control flow specifications in KARL, thus giving them an imperative flavor. An algorithm is built from primitive expressions which either assign a value to a variable or invoke an inference action (or another algorithm) and assign the return values to role variables. In the design phase, variables used in an algorithm may be of any of the data types available in DesignKARL. Primitive expressions may be combined using the control constructs sequence, iteration, and alternative.

In addition, *enriched inference actions* may be used as a preliminary form of algorithm. Enriched inference actions are described by logical clauses which, unlike inference actions during knowledge acquisition, also contain references to DesignKARL data types by means of their pre-defined test, retrieval, or storage operations. This type of description is intended particularly for the refinement of bodies of inference actions as a consequence of introducing data types. In the course of development, enriched inference actions may be further refined into algorithms. Note that the logic-based description of an enriched inference action must not comprise imperative-style primitives as used in the description of algorithms.

For example, after introducing an instance of the data type sequence to the store *part-method-pairs*, the inference action *Prune* (see Figure 5) can be turned into an algorithm which removes invalid associations from the sequence method-part-pairs rather than flagging them as being valid or invalid:

```
ALGORITHM Prune
        PREMISES conditions, part-method-pairs;
        CONCLUSIONS part-method-pairs;
                ...
                WHILE I <= LENGTH(part-method-pairs) DO
                        IF NOT check(part-method-pairs[I]) THEN
                                part-method-pairs := REMOVE(part-method-pairs, I);
                ...
END;
```

Algorithms may be introduced either in the context of methods associated to object classes at the domain and inference layer or in the bodies of inference actions. Algorithms may be subject to decomposition, i.e. may invoke more elementary algorithms as well as enriched inference actions. Algorithms can be introduced at any level of decomposition, i.e. they may substitute composed inference actions and the corresponding subtasks as well as refine elementary inference actions.

Typically, declarative specifications of elementary inference actions are refined to algorithms by adding control information while preserving validity. In this context, the fact that an algorithmic description yields a (not necessarily proper) subset of the output values of the original declarative specification given the same input data constitutes a sufficient condition for validity. For validity it is not always required to reproduce the set of output values of the original description exactly. If only a subset of the original output data is actually used for further processing it is sufficient for the validity of the algorithm to compute just this subset. Likewise, it is not mandatory to meet the above condition at each level of decomposition of the two representations.

Modules

Experience in conventional software engineering shows the need to reduce the complexity of

large software systems by decomposing them into manageable, largely self-contained pieces. To that end, DesignKARL introduces modules as structuring primitives. As the design phase should aim at preserving the structure of the KARL Model, DesignKARL provides two types of modules, namely *domain modules* and *processing modules*, which retain the distinction between domain-specific and domain-independent parts of the required knowledge. *Domain modules* collect related domain knowledge in a single place. A *processing module* corresponds to the decomposition of a composed inference action together with its associated subtask. Domain and processing modules can only interact in defined ways. More detailed accounts of the module concept of DesignKARL may be found in (Landes & Studer, 1994).

Language primitives for the process model

The language primitives that DesignKARL provides for the description of the design process itself are similar in spirit to proposals that have been made to express design rationale in a more general context (see Moran & Carroll (1996) for an overview of the field).

Goals and Their Relationships

The non-functional requirements modelled in the *NFR layer* (see section 3.3) constitute the basis for the design process. NFRs are viewed as goals to be achieved by means of suitable design decisions. Thus, we take a goal-oriented viewpoint on KBS design as, e.g., Dardenne et al. (1993) do on requirements engineering and Mylopoulos et al. (1992) do on information systems design.

The refinement of the functional specification, e.g. by using instances of particular data types, is motivated by NFRs and is effected by elementary design decisions (see below). Yet, top-level requirements formulated in the NFR layer tend to be fairly coarse: a top level goal might, e.g., express that “the system should be able to find a solution faster than a human expert” (cf. Figure 4). It is not immediately clear how such a requirement can be met or which portions of the system it affects specifically. Therefore, goals are gradually decomposed into subgoals until they can be satisfied by performing a collection of elementary design decisions. As decomposition continues, goals tend to become more constructive, i.e. they provide a high-level outline of how to reach a goal rather than just claim that a requirement must be met. Usually, goals can be decomposed in several ways, i.e. there are alternative ways to reach a goal. Therefore, decomposition of goals generally results in an AND/OR graph.

DesignKARL describes *GOALs* in an object-like fashion, i.e. it characterizes them by means of particular attributes. Attributes include, e.g., ancestors and successors in the decomposition hierarchy, dependencies on and correlations with other goals, the importance of the goal, its status (*ACTIVE*, *INACTIVE*, *ACHIEVED*, etc.), references to the evaluation criterion used as well as the portion of the design product affected by the goal, and a textual description of the goal.

The high-level requirement mentioned above can, for example be decomposed to a subgoal indicating that the assessment of possible *part-method* associations should be carried out efficiently, which in turn can be decomposed to a set of subgoals, one of which indicates that the most promising *part-method* associations should be retrievable quickly. In DesignKARL notation, the latter goal is expressed as:

```
GOAL RtEffPartMethPairAccess
        DECOMPOSITION OF RtEffPartMethPairAssessment;
        ACHIEVED BY STPartMethodPairs, ...;
        REFERENCE COMPOSED INFERENCE ACTION Assess,
                STORE part-method-pairs;
        STATUS ACTIVE;
```

DESCRIPTION "Make sure that those method-part associations can be retrieved most efficiently where the method involved incurs the least costs.";
IMPORTANCE HIGH;
END;

This goal also indicates that it will not be decomposed further, but can be achieved by applying several elementary design decisions, one of which (STPartMethodPairs) will be detailed below.

During the decomposition of goals, conflicts between goals may arise which must be resolved. Similar to the NFR context, claims that substantiate goals involved in a conflict are modelled as *ARGUMENTs*. *CONFLICTs* and *ARGUMENTs* are again DesignKARL primitives that are associated with suitable attributes.

Since goals may be decomposed in various ways, the designer has to select a subset of the available alternatives that seems to be most suitable in the given context. The motivation for preferring one alternative over another can be expressed by *PREFERENCEs* in a similar way as *ARGUMENTs* explain the reasons for resolving a conflict. *PREFERENCEs* indicate the criteria which makes an alternative preferrable over another or explicate the previous decisions on which the current decision depends.

In some cases, the selection of an alternative may be due to the fact that some potential alternatives are excluded because they are incompatible with previous design activities or, conversely, implied by earlier activities. These circumstances can be expressed as *IMPLICATIONs* or *EXCLUSIONs* between subgoals in the process model.

Often, design decisions are made tentatively and have to be withdrawn at a later stage of the design process when additional information has been gained. This circumstance can be expressed as a *REVISION* which points to the elements of the process model that are now superseded. Revised elements are retained in the process model since they document the design alternatives which have already been explored unsuccessfully and thus prevent a waste of resources by exploring them again. However, the status of these elements is changed to *INACTIVE* as a consequence of the revision.

Elementary Design Decisions

The decomposition of a goal continues until subgoals that can be achieved by means of elementary design decisions are reached . Elementary design decisions effect a modification of a part of the product model, i.e. the "refinement" of the functional specification. Thus, elementary design decisions also establish the link between functional requirements (embedded in the product model) and non-functional requirements (captured in the process model). Four basic types of elementary design decisions are distinguished in MIKE and provided as language primitives of DesignKARL: *REFINEMENT*, which refines parts of the model by introducing algorithms and data structures, *STRUCTURE*, which indicates the application of structuring primitives to decompose the overall model into smaller, largely self-contained parts or externally visible modifications of such parts, *INTRODUCTION*, which refers to portions of the model which appear without being a refinement of previously existing parts of the model, and *ELIMINATION*, which indicates that parts of the model are no longer needed and, thus, removed.

During the design process, the product model passes through a series of states which are determined by the versions of its constituents. The transition between two adjacent states is caused by (a collection of) elementary design decisions. The description of elementary design decisions in the process model indicates which parts of a state of the product model are replaced by new versions in the following state. Furthermore, *REFINEMENTs* also specify

by means of logical formulae the way in which the two versions of affected constituents relate to each other in detail. Elementary design decisions may also be undone as a consequence of revisions.

For instance, the introduction of an instance of the data type sequence in the store *part-method-pairs* is a refinement which contributes to the achievement of the goal *RtEffPart-MethPairAccess* mentioned above. This particular design decision can be specified in DesignKARL by expressing the fact that the elements of the sequence *part-method-pairs* in version z_1 are elements of the class *part-method-assoc* as in the previous version z_0 of the store in question, and that these elements are ordered by increasing cost incurred:

REFINEMENT STPartMethodPairs
PRE STORE part-method-pairs$@z_0$;
POST STORE part-method-pairs$@z_1$;
STATUS ACTIVE;
RULES
$(X = part\text{-}method\text{-}pairs[I] \wedge Y = part\text{-}method\text{-}pairs[J] \wedge I < J)@z_1 \rightarrow$
$(X \in part\text{-}method\text{-}assoc \wedge Y \in part\text{-}method\text{-}assoc \wedge$
$X[cost{:}\ C1] \wedge Y[cost{:}\ C2] \wedge C1 < C2)@z_0$.
END;

The model adopted in MIKE for describing the rationale of design decisions is in some parts based on earlier work by Potts & Bruns (1988) and Lee (1991) which promote an issue-based style, basically consisting of setting up questions and providing potential answers. In MIKE, a more result-oriented stance is taken and design decisions are linked to requirements more directly. It is not attempted to capture the discourse leading to the preference of one possible solution over others in order to avoid burdening the designer with too much additional overhead for documentation. Also, neither Potts & Bruns (1988) nor Lee (1991) provide equivalents to elementary design decisions in their schemes. They also do not address the elicitation and interpretation of requirements in their models.

4 THE MIKE TOOL

For supporting the development process of MIKE a tool called MIKE tool has been developed (Neubert, 1993). This tool has the following features:

- It provides structured editors for the knowledge protocols which include a text editor for the protocols itself and graphical editors to create and modify the additional structuring information of the knowledge protocols.
- Graphical editors support the creation and modification of the graphical primitives of the MoE in the Structure Model and the KARL Model. This includes an editor for program flow diagrams at the control layer, for the extended data flow diagrams at the inference layer, and the object diagrams at the domain layer. These graphical representations are easily understandable and provide a high and informal level for the representation of the knowledge. Thus they may be easily communicated to the expert and they may be used for documentation and maintenance.
- The MIKE-tool integrates an editor for the language KARL
- The testing of the KARL Model is supported by an interpreter for KARL. This interpreter is the basis for a debugger for KARL (see Figure 7). This debugger works on the graphical representation of the inference layer and provides breakpoints, inspection of stores,

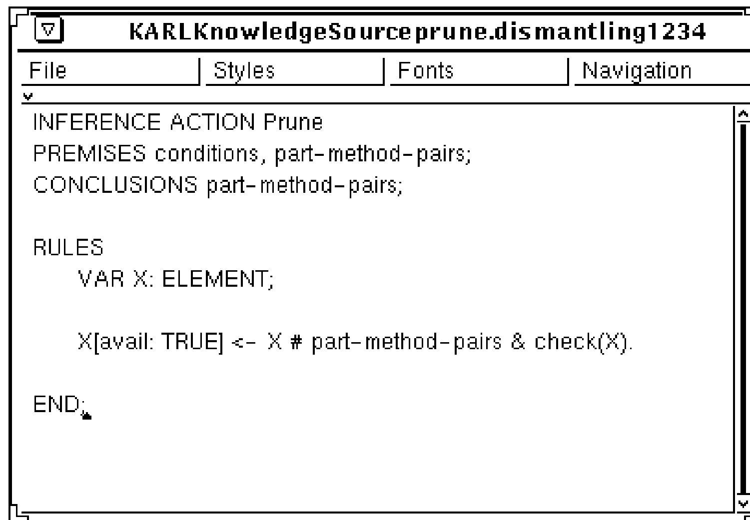

Figure 6 Editor of the MIKE-Tool for KARL. The picture shows the formal representation of the inference action *prune*.

step by step execution etc. This allows the KARL Model to be built by prototyping.

- The MIKE tool supports the transfer process between the different development steps by supporting the common conceptual model for the Structure Model, the KARL Model and the Design Model. Model part created in earlier steps may be enriched by additional knowledge in later steps. So for instance the semi-formal representation in the Structure Model is enriched by formal descriptions in KARL in the Formalization step.
- The MIKE tool automatically creates and administers the links between the different models. Thus traceability is supported between all the developed models.

Up to now this tool does not provide additional support for the development process itself, i.e. it does not survey and restrict the sequence of the development steps as described in section 2).

Similar tools for supporting the development process of KBS as well as for the development process of "conventional software" have been realized for other methods too.

The tools that were developed by commercial partners in the KADS-I and CommonKADS projects are basically customized editors using the structure of the semiformal model of expertise. However, neither the formalization of the model of expertise nor the development of the design model and in consequence no links between different modelling aspects are supported. Finally, no evaluation support is provided. Neither validation via executing a specification like in MIKE nor verification is available. Therefore, these tools only cover a small fragment of the tool support in MIKE.

The tool OMT-Select which supports OMT (Rumbaugh et al., 1991), provides graphical editors for state transition diagrams, data flow diagrams, and object diagrams. These diagrams correspond to the dynamic, functional, and static points of view of OMT-models. OMT-Select administers the relations between the three models by means of a common dictionary. The basic processes within the functional view are represented either by structured natural language or by an implementation. Therefore, such a model lacks an abstract formal description of the functionality. As a consequence, it only supports prototyping at the implementation level. In contrast to the MIKE tool it does not distinguish different models for different aspects but integrates analysis, design and implementation into one model.

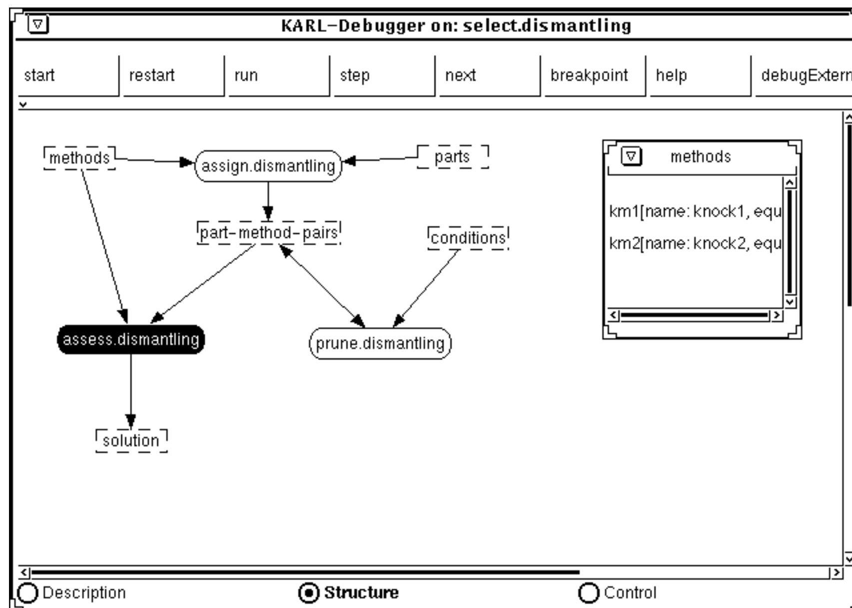

Figure 7 Graphical debugger of the MIKE-Tool for KARL. The current inference action is highlighted. The content of the store *methods* is shown in the small window.

5 RELATED WORK

MIKE relies heavily on the model of expertise as it was developed in the KADS-I (Wielinga et al., 1992) and CommonKADS (Schreiber et al., 1994) projects. The success of this model is based on the fact that it incorporates many of the existing approaches to knowledge engineering. Therefore, the use of this model makes MIKE comprehensible and enables sharing results with other approaches. However, some important differences exist when comparing MIKE with CommonKADS:

- In CommonKADS, the semiformal model in CML (Schreiber et al., 1994) is only loosely coupled with the formalization language $(ML)^2$ (van Harmelen & Balder, 1992). $(ML)^2$ uses different modeling primitives making the conceptual distance between the semiformal and formal models much wider. Conversely, a formal specification in KARL is a refinement of the semiformal model by adding an additional level of detail.[*]
- MIKE includes prototyping into a model-based framework. The executability of KARL allows models to be evaluated in an early stage of the development process which prevents high costs in detecting and revising errors during the implementation phase.
- CommonKADS makes a clear and sharp distinction between several models used to represent the development process of a KBS. MIKE provides a smooth transition between its model types. Actually, these models are different levels of refinement of the same model pattern, the MoE. Based on the smooth transitions, MIKE provides strong support in tracing development decisions through the cascade of models supporting revision and maintenance of the KBS.
- Finally, CommonKADS does not provide a systematic treatment of design issues compa-

*. Fensel & van Harmelen (1994) provide a detailed comparison of KARL and $(ML)^2$.

rable to the Design Model and DesignKARL in MIKE.

Similar to MIKE Structured Analysis (Yourdon, 1989) and OMT (Rumbaugh et al., 1991) distinguishes different points of view during analysis. A static view is established by an entity-relationship model and the data dictionary in Structured Analysis or by object diagrams in OMT. The function oriented point of view considers the data flows between the processes. These processes are hierarchically refined. Elementary processes are described semi-formally in a pseudo-code notation. This point of view strongly resembles the inference layer of the MoE. State-transition diagrams represent the control flow in OMT which thus correspond to the control layer in MIKE. Due to the semi-formal nature of these models they correspond to the semi-formal variant of the MoE: the Structure Model. The Design Model of MIKE corresponds to the result of Structured Design or the design model of OMT. Despite these commonalities a lot of differences exist:

- The separation of knowledge about the domain and knowledge about the problem-solving method has no analogy in Structured Analysis or OMT. Here both aspects are mixed which prevents the independent reuse of both parts.
- Neither OMT nor Structured Analysis provides a formal variant of their system model. Thus this model still allows room for interpretation and is thus not an unambiguous document for the further development process.
- As a consequence of the previous point, the system models of Structured Analysis and OMT are not executable. These models may not be evaluated by executing them and thus they may not be built by prototyping within the analysis phase.
- The successor of the system model is the design model in Structured Analysis which may be structured very differently. Additionally, no explicit links are supported between the two models. This complicates traceability of the models and makes maintenance and revision much more difficult.

The construction and reuse of models as part of requirements engineering has attracted a lot of interest in recent years (see e.g. Jarke et al., 1993). Sutcliff & Maider, (1994) introduce the notion of object system models which is used for describing types of applications like object composition (which can be instantiated to manufacturing systems) or agent-object control (which can be instantiated to command and control applications). Such object system models are embedded into a specialization hierarchy: each object type may be refined to obtain further specialized object system models by using additional knowledge types for discrimination. These object system types provide a framework for reuse during requirements engineering since they can be used as initial generic descriptions of application types. Compared to the notion of reusable problem-solving methods, object system models are used rather for describing problem spaces whereas problem-solving methods are used for specifying solution spaces (compare Sutcliff & Maiden, 1994). It should be noted further noticed that the notion of domain model as used in (Sutcliff & Maiden, 1994) has a much broader meaning than the notion of domain model as used in the KADS or MIKE framework, since behavioral aspects are not included in the domain model there, these aspects are modeled rather at the inference and control layers of the Model of Expertise.

6 CONCLUSION

MIKE integrates semiformal, formal, and operational description formalisms in an incremental development process. The semiformal Structure Model is not only used to facilitate the formalization process, but is also seen as an important result itself. It structures the domain knowledge and the knowledge about the problem-solving process (task related knowledge) and due to its semi-formal description it can be used for documentation. The formal specifi-

cation describes the functionality of the system precisely, yet abstracting from implementation details. Since the formal specification is operational, it is used as a prototype to validate the Model of Expertise. The clear separation of knowledge about the problem-solving method and domain knowledge allows the reuse of these parts. During design, the formal specification is extended with respect to aspects related to the implementation of the system, taking non-functional requirements into particular account.

Due to the common underlying conceptual model, the different representations can easily be linked to each other and there is a smooth transition from one representation to the other. By linking the models, we gain the advantage of using, e.g., the semiformal model as an additional documentation of the formal specification. Furthermore, requirements traceability is supported by interrelating all the models. The MIKE tool environment provides different integrated graphical editors for constructing the models and their relationships and it integrates a debugging tool based on the interpreter for KARL.

In that way MIKE addresses one of the main topics which have been put on the research agenda for requirements engineering in software engineering and information systems engineering: combination of different representations (Pohl et al., 1995) based on a strong conceptual model involving aspects like smooth coupling of different representations, traceability and consistency.

In the meantime MIKE has been applied to more than a dozen applications including the mentioned application for selecting techniques for dismantling buildings (Fichtner et al., 1995), an application for supporting a help desk service for users of a communication network (Waarle, 1995), an application for designing artefacts in engineering tasks (Poeck et al., 1996), and an application for determining optimal shift-systems for various types of companies (Gissel & Knauth, 1997).

Current work addresses the verification of formal specifications (Fensel et al., to appear (b); Fensel & Schönegge, 1997); extensions of the Model of Expertise to cope more adequately with issues of reuse (Angele et al., 1996b; Fensel & Groenboom, 1997); extensions of MIKE to address organisational and business process aspects (Decker et al., 1997); and the development of broker architectures that enable use and reuse of ontologies and problem-solving methods in distributed networks like the WWW (Fensel et al., 1997; Fensel, 1997a).

Acknowledgement:

We thank Susanne Neubert who provided valuable contributions to many of the ideas addressed in this paper and Wolf Fichtner who developed the MIKE solution for the dismantling task. The support of the Institute of Industrial Production (IIP) of the University of Karlsruhe in handling the dismantling tasks is gratefully acknowledged.

7 REFERENCES

Angele, 1993J. Angele (1993): Operationalisierung des Modells der Expertise mit KARL, PhD thesis University of Karlsruhe, Infix-Verlag, St. Augustin, 1993.

Angele, 1996J. Angele (1996): Conceptual Modeling in KARL and G-KARL. In *Proceedings of the CASE Workshop during the 15th International Conference on Conceptual Modelling (ER-96)*, Cottbus, October 7-10, 1996.

Angele et al., 1996aJ. Angele, D. Fensel, and R. Studer (1996): Domain and Task Modelling in MIKE. In A. Sutcliffe et al. (eds.): *Domain Knowledge for Interactive System Design*, Chapman & Hall, 1996.

Angele et al., 1996bJ. Angele, S. Decker, R. Perkuhn und R. Studer (1996): Modeling Problem-Solving Methods in New KARL. In *Proceedings of the 10th Knowledge Acquisition for Knowledge-Based Systems Workshop (KAW´96)*, Banff, Canada, November 1996.

Benjamins et al., 1996 R. Benjamins, D. Fensel, and R. Straatman (1996): Assumptions of Problem-Solving Methods and Their Role in Knowledge Engineering. In *Proceedings of the 12. European Conference on Artificial Intelligence (ECAI-96)*, Budapest, August 12-16, 1996.

Boehm, 1988B.W. Boehm (1988): A Spiral Model of Software Development and Enhancement, *IEEE Computer,* May 1988, 61-72.

Breuker & Van de Velde, 1994J. Breuker and W. Van de Velde (eds.) (1994): *The CommonKADS Library for Expertise Modelling*. IOS Press, Amsterdam, The Netherlands, 1994.

Chandrasekaran et al., 1992B. Chandrasekaran, T.R. Johnson, and J. W. Smith (1992): Task Structure Analysis for Knowledge Modeling, *Communications of the ACM*, 35(9), 1992, 124—137.

Clancey, 1983W. J. Clancey (1983): The Epistemology of a Rule-Based Expert System—a Framework for Explanation, *Artificial Intelligence*, 20, 1983, 215—251.

Dardenne et al., 1993A. Dardenne, A. van Lamsweerde, and S. Fickas (1993): Goal-directed requirements acquisition, *Science of Computer Programming 20*, 1993, 3-50.

Decker et al., 1997S. Decker, M. Daniel, M. Erdmann, and R. Studer (1997): An Enterprise Reference Scheme for Integrating Model-based Knowledge Engineering and Enterprise Modeling. In *Proceedings of the 10th European Workshop on Knowledge Acquisition , Modeling, and Management (EKAW-97)*, LNAI, Springer Verlag, 1997.

de Kleer et al., 1992J. de Kleer, A. K. Mackworth, and R. Reiter (1992): Characterizing Diagnoses and Systems, *Artificial Intelligence*, 56, 1992.

Eriksson, 1992H. Eriksson (1992): A survey of knowledge acquisition techniques and tools and their relationship to software engineering, *Journal of Systems and Software* ,19: 97-107, 1992.

Farquhar et al., 1997A. Farquhar, R. Fikes, and J. Rice: The Ontolingua Server (1997): a Tool for Collaborative Ontology Construction, *International Journal of Human-Computer Studies (IJHCS)*, 46(6), 1997, 707—728.

Feigenbaum, 1977E. A. Feigenbaum (1977): The Art of Artificial Intelligence: Themes and Case Studies of Knowledge Engineering. In *Proceedings of the International Joint Conference on AI (IJCAI-77)*, 1977, 1014—1029.

Fensel, 1995D. Fensel: *The Knowledge Acquisition and Representation Language KARL.* Kluwer Academic Publ., Boston, 1995.

Fensel, 1997aD. Fensel (1997a): An Ontology-based Broker: Making Problem-Solving Method Reuse Work. In *Proceedings of the Workshop on Problem-Solving Methods for Knowledge-based Systems at the 15th International Joint Conference on AI (IJCAI-97)*, Nagoya, Japan, August 23, 1997.

Fensel, 1997bD. Fensel (1997b): The Tower-of-Adapter Method for Developing and Reusing Problem-Solving Methods. To appear in *Proceedings of European Knowledge Acquisition Workshop (EKAW-97)*, LNAI, Springer-Verlag, 1997.

Fensel & Groenboom, 1997D. Fensel and R. Groenboom (1997): Specifying Knowledge-Based Systems with Reusable Components. In *Proceedings of the 9th International*

Conference on Software Engineering & Knowledge Engineering (SEKE-97), Madrid, Spain, June 18-20, 1997.

Fensel & Schönegge, 1997 D. Fensel and A. Schönegge (1997): Specifying and Verifying Knowledge-Based Systems with KIV. In *Proceedings of the European Symposium on the Validation and Verification of Knowledge Based Systems EUROVAV-97*, Leuven Belgium, June 26-28, 1997.

Fensel & Straatman, 1996D. Fensel and R. Straatman (1996): The Essence of Problem-Solving Methods: Making Assumptions for Efficiency Reasons. In N. Shadbolt et al. (eds.): *Advances in Knowledge Acquisition,* LNAI 1076, Springer-Verlag, 1996.

Fensel & van Harmelen, 1994D. Fensel and F. van Harmelen (1994): A Comparison of Languages which Operationalize and Formalize KADS Models of Expertise, *The Knowledge Engineering Review*, 9(2), 1994.

Fensel et al., 1993D. Fensel, J. Angele, D. Landes, and R. Studer (1993): Giving Structured Analysis Techniques a Formal and Operational Semantics with KARL. In Züllighoven et al. (eds): *Requirements Engineering '93: Prototyping*, Teubner Verlag, Stuttgart, 1993.

Fensel et al., 1997D. Fensel, M. Erdmann, and R. Studer (1997): Ontology Groups: Semantically Enriched Subnets of the WWW. In *Proceedings of the International Workshop Intelligent Information Integration during the 21st German Annual Conference on Artificial Intelligence*, Freiburg, Germany, September 9-12, 1997.

Fensel et al., to appear (a) D. Fensel, J. Angele, R. Studer (a): The Knowledge Acquisition and Representation Language KARL, to appear in *IEEE Transactions on Knowledge and Data Engineering*.

Fensel et al., to appear (b)D. Fensel, R. Groenboom and G. R. Renardel de Lavalette (b): MCL: Specifying the Reasoning of Knowledge-based Systems, to appear in *Data and Knowledge Engineering (DKE).*

Fichtner et al., 1995W. Fichtner, D. Landes, Th. Spengler, M. Ruch, O. Rentz, and R. Studer (1995): Der MIKE Ansatz zur Modellierung von Expertenwissen im Umweltbereich - dargestellt am Beispiel des Bauschuttrecyclings. In H. Kremers et al. (eds.): *Space and Time in Environmental Information Systems, Proceedings of the 9th Int. Symposium on Computer Science for Environmental Protection*, Berlin, September 1995, Metropolis Verlag, 1995.

Gennari et al., 1994J. Gennari, S. Tu, Th. Rothenfluh, and M. Musen (1994): Mapping Domains to Methods in Support of Reuse, *Int. J. of Human-Computer Studies (IJHCS)*, 41, 1994, 399-424.

Gissel & Knauth, 1997Gissel, A. and Knauth, P. (1997): Knowledge based support for the participatory design and implementation of shift systems, *Shiftwork International Newsletter* (14), 1, 9, 1997.

Gruber, 1993T.R. Gruber (1993): A Translation Approach to Portable Ontology Specifications, *Knowledge Acquisition 5(2):*199-221, 1993.

Hayes-Roth et al., 1983F. Hayes-Roth, D. A. Waterman, and D. B. Lenat (eds.) (1983): *Building Expert Systems*. Addison-Wesley Publisher, 1983.

Harel, 1984D. Harel: Dynamic Logic. In D. Gabby et al. (eds.) (1984), *Handbook of Philosophical Logic, vol. II, Extensions of Classical Logic*. Publishing Company, Dordrecht (NL), 1984, 497-604.

Jarke et al., 1993M. Jarke, J. Bubenko, C. Rolland, A. Sutcliffe, and Y. Vassiliou (1993): Theories Underlying Requirements Engineering: An Overview of NATURE at Genesis.

In *Proc. IEEE Symposium on Requirements Engineering*, San Diego, 1993.

Knight & Luk, 1994K. Knight and S. Luk (1994): Building a Large Knowledge Base for Machine Translation. In *Proc. of the American Ass. for Artificial Intelligence Conference (AAAI'94)*, Seattle, 1994.

Landes, 1994 D. Landes (1994): DesignKARL - A Language for the Design of Knowledge-Based Systems. In *Proceedings of the 6th International Conference on Software Engineering and Knowledge Engineering SEKE'94*, Jurmala, Latvia, June 20-23, 1994.

Landes & Studer, 1994D. Landes and R. Studer (1994): Mechanisms for Structuring Knowledge-Based Systems. In *Database and Expert System Applications*, D. Karagiannis, ed. Lecture Notes in Computer Science 856, Springer, Berlin, 1994, 488-497

Landes & Studer, 1995D. Landes and R. Studer (1995): The Treatment of Non-Functional Requirements in MIKE. In Wilhelm Schäfer, Pere Botella (eds.): *Proceedings of the 5th European Software Engineering Conference ESEC´95* (Sitges, Spain, September 25-28, 1995), Berlin, Springer, (Lecture Notes in Computer Science; 989), 1995.

Lenat & Guha, 1990D.B. Lenat and R.V. Guha (1990): *Building Large Knowledge-Based Systems: Representation and Inference in the CYC Project*. Addison-Wesley Publ. Co., Inc., Reading, Massachusetts, 1990.

Lee, 1991J. Lee (1991): Extending the Potts and Bruns model for recording design rationale. In *Proc. 13th Int. Conference on Software Engineering ICSE'91* (Austin, USA), 1991, 114-125.

Moran & Carroll, 1996T.P. Moran and J.M. Carroll (1996): *Design Rationale - Concepts, Techniques, and Use*. Erlbaum, Mahwah, 1996.

Morik, 1987K. Morik (1987): Sloppy Modeling. In K. Morik (ed.), *Knowledge Representation and Organisation in Machine Learning*, LNAI 347, Springer-Verlag, 1987.

Mylopoulos et al. 1992J. Mylopoulos, L. Chung, and B. Nixon (1992): Representing and using non-functional requirements: a process-oriented approach. In IEEE Transactions on Software Engineering 18(6), 1992, 483-497.

Neubert, 1993S. Neubert: Model Construction in MIKE (Model-Based and Incremental Knowledge Engineering). In N. Aussenac et al. (eds.), *Knowledge Acquisition for Knowledge-Based Systems, Proceedings of the 7th European Workshop (EKAW´93),* Toulouse, France, 1993, LNAI 723, Springer-Verlag, 1993.

Nebel, 1996 B. Nebel (1996): Artificial Intelligence: A Computational Perspective. In G. Brewka (ed.): *Principles of Knowledge Representation*, CSLI publications, Studies in Logic, Language and Information, Stanford, 1996.

Newell, 1982A. Newell (1982): The Knowledge Level, *Artificial Intelligence,* 18:87-127, 1982.

Pirlein & Studer, 1994T. Pirlein and R. Studer (1994): KARO: An Integrated Environment for Reusing Ontologies. In Steels et al. (eds): *A Future of Knowledge Acquisition, Proc. 8th European Knowledge Acquisition Workshop (EKAW´94)*, Hoegaarden, LNCS, 867, Springer, 1994.

Pirlein & Studer, 1997Th. Pirlein and R. Studer (1997): Integrating the Reuse of Commonsense Ontologies and Problem-Solving Methods, University of Karlsruhe, Institute AIFB, Research Report 354, May 1997, submitted for publication.

Poeck et al., 1996K. Poeck, D. Fensel, D. Landes, and J. Angele (1996): Combining KARL And CRLM For Designing Vertical Transportation Systems, *The International Journal of Human-Computer Studies (IJHCS)*, 44(3-4), 1996.

Pohl et al., 1995 K. Pohl, G. Starke, and P. Peters (1995): Workshop Summary First International Workshop on Requirements Engineering: Foundation of Software Quality (REFSQ´94), *ACM SIGSOFT,* 20(1), 1995, 39—45.

Potts & Bruns, 1988C. Potts and G. Bruns (1988): Recording the Reasons for Design Decisions. In *Proc. 10th Int. Conference on Software Engineering ICSE'88* (Singapore), 1988, 418-427.

Przymusinski, 1988T. C. Przymusinski (1988): On the Declarative Semantics of Deductive Databases and Logic Programs. In J. Minker (ed.): *Foundations of Deductive Databases and Logic Programming*, Morgan Kaufmann Publisher, Los Altos, CA, 1988.

Puppe, 1993F. Puppe (1993): *Systematic Introduction to Expert Systems: Knowledge Representation and Problem-Solving Methods*. Springer, 1993.

Rumbaugh et al., 1991J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy und W. Lorensen (1991): *Object-Oriented Modeling and Design*. Prentice Hall, 1991.

Schreiber, 1993G. Schreiber (1993): Operationalizing models of expertise. In (Schreiber at al., 1993), 119-149.

Schreiber at al., 1993G. Schreiber, B. Wielinga, and J. Breuker (eds.) (1993): *KADS - A Principled Approach to Knowledge-Based Systems Development*. Academic Press, London, 1993.

Schreiber et al., 1994 A. TH. Schreiber, B. Wielinga, J. M. Akkermans, W. Van De Velde, and R. de Hoog (1994): CommonKADS. A Comprehensive Methodology for KBS Development, *IEEE Expert*, 9(6), 1994, 28—37.

Studer et al., 1996 R. Studer, H. Eriksson, J. Gennari, S. Tu, D. Fensel, and M. Musen (1996): Ontologies and the Configuration of Prolem-Solving Methods. In *Proc. of the 10th Knowledge Acquisition for Knowledge-based Systems Workshop (KAW-96)*, Banff, November 1996.

Sutcliff & Maiden, 1994A.G. Sutcliff and N.A.M. Maiden (1994): Domain Modeling for Reuse. In *Proc. 3rd International Conference on Software Reuse*, Rio de Janeiro, 1994.

Ullman, 1988J. D. Ullman (1988): *Principles of Database and Knowledge-Base Systems, vol I*. Computer Sciences Press, Rockville, Maryland, 1988.

van Harmelen & Balder, 1992F. v. Harmelen and J. Balder: $(ML)^2$: A Formal Language for KADS Conceptual Models, *Knowledge Acquisition*, vol 4, no 1, 1992.

van Heijst et al., 1997 G. van Heijst, A. T. Schreiber, and B. J. Wielinga (1997): Using Explicit Ontologies in Knowledge-Based System Development, *International Journal of Human-Computer Interaction (IJHCI)*, 46(6), 1997.

Waarle, 1995H. Waarle (1995): Knowledge-based System Modeling Using MIKE. Swiss PTT, R&D Department, Technical Report, November 1995.

Wielinga & Breuker, 1984 B. J. Wielinga and J. A. Breuker (1984): Interpretation Models for Knowledge Acquisition. In *Proceedings of the European Conference on AI (ECAI-84)*, Pisa, 1984.

Wielinga et al., 1992Wielinga, B., Schreiber, G. & Breuker, J., A. (1992): KADS: A Modeling Approach to Knowledge Engineering, *Knowledge Acquisition*, 4(1), 1992.

VanLehn, 1989K. VanLehn (1989): Problem-Solving and Cognitive Skill Acquisition. In M. I. Posner (ed.): *Foundations of Cognitive Science*, The MIT Press, Cambridge, 1989.

Yourdon, 1989E. Yourdon (1989): *Modern Structured Analysis*. Prentice-Hall, Englewood Cliffs, 1989.