The Design Process in MIKE

Dieter Landes and Rudi Studer

Institut für Angewandte Informatik und Formale Beschreibungsverfahren
Universität Karlsruhe, D-76128 Karlsruhe, Germany

e-mail: {landes | studer}@aifb.uni-karlsruhe.de

Abstract. Not only knowledge acquisition, but also the principled transition from knowledge acquisition to implementation is an important question in knowledge engineering. This paper focuses on this transition and characterises the design phase of the MIKE approach to model-based incremental development of knowledge-based systems. The role of non-functional requirements in the development process is discussed and it is argued that for reasons of transparency, the description of the design should not just contain the description of the artefact, but should additionally include a record of the design decisions and their motivation, i.e. their relationship to the requirements posed. DesignKARL is outlined as the formalism for describing the artefact as well as design decisions and their rationale in MIKE.

1 INTRODUCTION

In recent years, it has been recognized that building expert systems according to the rapid prototyping paradigm comes along with a number of shortcomings (cf., e.g., (Angele and Fensel, 1992)). In particular, the knowledge engineer has to tackle a variety of tasks in a completely intertwined fashion, i.e. she must simultaneously analyse the given information, design, implement, and evaluate the system. Consequently, different aspects of knowledge must be considered in parallel and will often be mixed up in the implementation. As another drawback, the final implementation often constitutes the only documentation of the expertise incorporated in the system.

In order to overcome these problems for being able to build large-scale systems which can be used and maintained over a long period of time, more principled methods for constructing knowledge-based systems (kbs) are required. As one such principled framework, the MIKE (Model-based Incremental Knowledge Engineering) approach has been proposed. MIKE shares many of the basic ideas with KADS (Wielinga, Schreiber, and Breuker, 1993; Schreiber, Wielinga, and Breuker, 1993) and aims at integrating the benefits of life-cycle models, prototyping, and formal specifications in a single method (Angele, Fensel, Landes, Neubert, and Studer, 1993).

The focus of this paper lies on the design phase of the MIKE life-cycle and the design model as the result of this phase. The basic principles of MIKE will briefly be addressed in section 2 in order to clarify the context of the design model. The role of non-functional requirements and other aspects that influence shape and contents of the design model are discussed in section 3. Section 4 outlines a formalism for the design model which is capable of describing the artefact under construction as well as design decisions and their rationale. Section 5 puts the work presented here in relation to some other approaches, and section 6 concludes the paper with a discussion of the approach.

2 SOME CORE IDEAS OF MIKE

In (Angele, Fensel, Landes, Neubert, and Studer, 1993), it has been elaborated that knowledge engineering has to be viewed as a modelling process which involves interpretation of elicited knowledge and observations of the expert by the knowledge engineer. Since interpretation is by its very nature prone to errors and misunderstandings, early feedback by reality is required and revisions of the model must be admissible in any stage of the development process. Furthermore, as even the part of reality which is relevant for an expert task is often too complex to be covered completely by a model, such a model is only an approximation which is subject to further refinement or modification due to new observations. Thus, knowledge engineering must be carried out incrementally.

The gap between informal knowledge as it is gained from a human expert and its final implementation is usually much too wide to be bridged in a single step. Experience in "conventional" software engineering shows that the development process should be split into several steps each of which concentrates on a particular aspect of the overall development task. Consequently, different representations are required each of which is particularly suited for describing the results of one of the development steps or substeps thereof. MIKE distinguishes *knowledge acquisition*, *design*, *implementation*, and *evaluation* as basic development phases. As first substep of the knowledge acquisition phase, a natural language description of the expertise is constructed. This description is the basis for the interpretation of the acquired knowledge which results in a semi-formal, but structured description by means of the so-called hyper model (Neubert, 1993). Interpretation is followed by formalization, resulting in a so-called model of expertise. Its shape in MIKE is strongly influenced by ideas from KADS in that three basic types of knowledge are distinguished, which are represented at different layers of the model of expertise. The representation formalism for the model of expertise in MIKE is the formal specification language KARL (Fensel, 1993; Angele, Fensel, and Studer, 1994). Since KARL possesses an operational semantics (Angele, 1993) in addition to its model-theoretic semantics, the model of expertise can be evaluated by means of (basically) explorative prototyping (Floyd, 1984). The tight integration of prototyping into the development process is a consequence of the iterative and approximative nature of that process.

Models in the knowledge acquisition phase focus on how the task at hand can be solved, but abstract from considerations how this solution will finally be realized on a computer. The latter issues are addressed in the design phase and described in its result, the design model. The design model then contains an account of all the requirements the system has to meet and how these goals can be achieved. The design model serves as the basis for implementing the requirements in the final system which is the result of the implementation phase.

Finally, the implemented system has to be evaluated in order to determine if the requirements posed are actually met by the system in its entirety.

As knowledge engineering is an incremental process, these phases are traversed in a cyclic fashion similar to Boehm's spiral model (Boehm 1988) (see Figure 1). Details on the life-cycle of MIKE can be found in (Angele, Fensel, Landes, Neubert, and Studer, 1993; Angele, 1993).

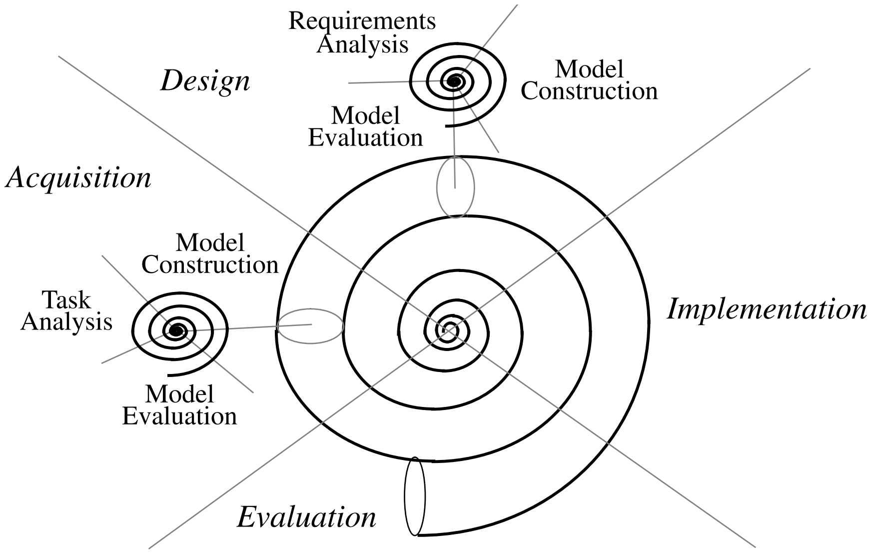

Figure 1 The knowledge engineering process in MIKE (reprinted from (Angele, Fensel, Landes, Neubert, and Studer, 1993))

3 THE SCOPE OF THE DESIGN MODEL

3.1 Two Basic Types of Requirements

In life-cycle oriented approaches to software development, the analysis phase usually emphasizes a certain portion of requirements, namely the *functional* aspects of the problem's solution. In addition to functional requirements, *non-functional* requirements must also be captured, described, and finally met by the realization of the system. (Kahn, Keller, and Panara, 1990) provide a collection of types of non-functional requirements (quality attributes in their terminology) such as efficiency, reliability, maintainability, etc. which are important for the quality of a software system in general. Compared to functional requirements, such requirements convey a different type of information: while functional requirements express which functionality the system must supply, non-functional requirements constrain how behaviour may be exhibited, i.e. non-functional requirements are in essence requirements *about* functional requirements.

The analysis phase in "conventional" software engineering approaches corresponds to the knowledge acquisition phase in knowledge engineering approaches. In contrast to specifications in "conventional" software engineering, the model of expertise as the result of knowledge acquisition cannot only address *what* functionality the system must provide, but also *how* the required functionality can be exhibited since knowledge about how to solve a problem constitutes an indispensable part of the expertise. Aspects concerning the *realization* of the functionality, however, are the concern of the design phase and neglected during knowledge acquisition. In MIKE, knowledge acquisition results in an executable description of the model of expertise which can be run as a prototype. Thus, this prototype constitutes a computational solution which is capable of solving the considered task at least in principle, though usually in an inefficient way. Consequently, the model of expertise can already be evaluated with respect to functional requirements before the transition into the design phase takes place (cf. also (Landes, 1993)). Thus, there is a clear distinction between the types of requirements to be addressed during knowledge acquisition on the one hand and design on the other hand.

What are the requirements that must be considered during the design phase of a kbs? While there is some work focusing particularly on non-functional requirements in other areas such as information system design (Chung, 1991; Mylopoulos, Chung, and Nixon, 1992), the investigation of relevant types of requirements for knowledge-based systems has become a research topic only recently. (Williams, Tomlinson, Bright, and Rajan, 1992) and (Guida and Mauri, 1993) provide taxonomies of factors that determine the quality of knowledge-based systems, but do not distinguish functional and non-functional aspects. Based on these taxonomies, *efficiency*, *maintainability*, *understandability*, *reliability*, *portability*, and *requirements resulting from the chosen hardware or software environment or the system architecture* have been identified so far as important top-level non-functional requirements in the design phase in MIKE. It should be noted that two aspects of efficiency have to be distinguished, namely efficiency as integral part of the problem-solving method and efficiency of the realisation of the problem-solving method. These two aspects have already been discussed as efficiency at the knowledge level vs. efficiency at the symbol level by (Schreiber, Akkermans, and Wielinga, 1990). While the former aspect has to be addressed during knowledge acquisition, the latter aspect is part of the design phase.

It should also be mentioned that MIKE currently concentrates on the development of reasoning mechanisms and knowledge base. Other constituents of a kbs such as user interface, data management component, etc. are, in spite of their importance for the success of the system, not addressed in depth at this stage of the project since issues concerning the development of these constituents are already much better understood through work in conventional software engineering. Thus, requirements referring to those constituents in particular are presently not captured in the design model. Due to their interdependencies, the development of the individual constituents of a kbs should nevertheless be carried out in parallel.

3.2 Where Do Non-Functional Requirements Come From?

Because of their specific character, non-functional requirements usually cannot be acquired completely during knowledge acquisition. Some indications how functionality has to be exhibited may be contained in informal documents of the hyper model (Neubert, 1993), e.g., hints about which results are needed with a certain precision or within a particular response time. Yet, knowledge acquisition focuses on conceptual issues such as identifying the activities to perform for solving a particular problem and getting hold of the knowledge required for these activities. This type of information is available in some form or the other from human experts for the particular task, whereas these experts usually have only very limited ideas about specific aspects of a computational solution besides that it must solve the task. Therefore, the main sources of information about non-functional requirements are future users of the system and the client, who wants the respective task performed by a computer. The clarification of which non-functional requirements have to be satisfied, but also the estimation to which extent known non-functional requirements are met can be supported by running a prototype which already conforms to (most of) the functional requirements. Emphasis, however, shifts from checking the functionality of the system (“doing the right things”) to evaluating how the system exhibits particular facets of its functionality (“doing things right”), i.e. prototyping gets an experimental flavour (Floyd, 1984).

A knowledge-based system usually encompasses additional components such as, e.g., a sophisticated user interface and an explanation facility besides the reasoning component. Since all the components co-operate during the use of the system, specific needs of one component may have repercussions on others. For instance, good explanations of the system's reasoning behaviour may require to store a larger amount of intermediate data or store data in a more explicit fashion than is required for just finding a solution. These needs constitute additional non-functional requirements for the reasoning component since they do not directly affect its functionality.

3.3 Characterization of the Design Process

3.3.1 Design Issues

The main points of interests in the design phase are the structure of the system and its constituents on the one hand, and data structures and algorithms on the other hand.

Although the co-operation of the components of a kbs may give rise to mutual requirements, many of a component's internals are irrelevant for others. Independence with respect to internals, however, can only be achieved if a suitable system structure and appropriate interfaces for the interaction between components are defined. In that case, the development of the components can even be carried out according to different development paradigms which are particularly suitable for the component in question. Furthermore, a proper system structure reduces the overall complexity of the development task. The structure of the system may also be influenced by requirements like maintainability or understandability.

As (Schreiber, 1993) points out, the design process should preserve the structure of the model of expertise since otherwise the model of expertise may, e.g., only provide limited support for maintenance. Therefore, structural modifications beyond refining existing structures should be employed with great care. Clearly, restructuring is unavoidable if the target software environment is not compatible with the development paradigm followed so far. For instance, structure preserving design is hard to accomplish if knowledge acquisition is carried out according to the MIKE proposal, i.e. pursuing a function-oriented (structured) approach in combination with object-oriented data modelling, but the target environment is a rule-based expert system shell. In that case, design can either preserve the structure of the model of expertise with its specification of control on a global level, thus leaving a wide gap between design and implementation, or design introduces structures which can directly be mapped to the implementation environment, thus causing structural changes with respect to the model of expertise since, e.g, the global control knowledge of the task layer has to be distributed among many individual rules. In order to avoid these problems, our considerations are currently restricted to target software environments which allow for structure-preserving design such as, e.g., C++.

The second basic design aspect besides system structure is system behaviour, which can be influenced by the choice of data structures and algorithms. For instance, data management may be improved by appropriate data structures while processing may be improved by employing more sophisticated algorithms. Changes in data structures usually imply that algorithms using them have to be adapted correspondingly. Conversely, a particular algorithm mostly requires data structures which are tailored to its needs. It should be made sure that decisions taken in this context actually constitute *design* decisions: if decisions significantly affect the functionality of the system, e.g., by resulting in a different problem-solving method, a new iteration of knowledge acquisition should be carried out, thus propagating the changes also into design, instead of just modifying the design model and leaving the model of expertise intact. The original model of expertise as it results from the most recent iteration of the knowledge acquisition phase is preserved unchanged during design.

3.3.2 Addressing Design Issues

The design phase consists of three subphases, namely *requirements analysis*, *model construction*, and *model evaluation*, which are traversed in a cyclic fashion. Figure 1 at page 3 shows these steps in the context of the MIKE life-cycle.

Requirements analysis involves collecting comments of clients and users concerning the current realization of the system. Such comments are recorded, analysed, and described as non-functional requirements. Thus, one basic activity in the analysis substep is the identification of additional, previously unnoticed non-functional requirements that the system has to meet. On the other hand, given a collection of non-functional requirements, the designer usually cannot address them all at once; rather, only a subset will be tackled before aiming at the remaining ones. Thus, a second aspect in the analysis step consists in analysing the non-functional requirements in order to select those to address next, e.g., those currently associated with the highest risk.

After the selection of a subset of the requirements, the designer decides on possible ways to meet the selected requirements or at least to contribute to their fulfilment. Such design activities imply changes of portions of the artefact. Thus, the model construction step consists in performing suitable design activities which result in a new “version” of the artefact. The decision process can be viewed as first posing a top-level goal, namely to reach a state in which the model of the system meets the chosen non-functional requirements. Since top-level goals usually cannot be met immediately, they will be decomposed into subgoals (cf., e.g., (Mylopoulos, Chung, and Nixon, 1992)). Some of these subgoals must be fulfilled jointly to satisfy a higher-level goal while others constitute alternatives such that the higher-level goal is fulfilled if any of the subgoals is met. Goal decomposition is continued until elementary goals are reached which can be directly achieved by elementary design decisions such as, e.g., refining a particular data structure in a specific way.

Finally, design decisions have to be *evaluated* if they actually effected an improvement with respect to the selected requirements under the constraint that the system still exhibits the required functionality. Valuable support for evaluation can be given by running an operational prototype of the system which reflects the effects of design decisions. The result of the evaluation is the basis for the next requirements analysis step within the design phase. If decisions taken turn out as failure, the chosen requirements must be re-considered and the decisions just taken must be revised. It is also possible that the decisions taken are initial steps towards the satisfaction of the chosen requirement, but do not yet suffice to fulfil it to the desired extent. In that case, complementary decisions and activities are required. Finally, if the result of evaluation is satisfactory, the focus shifts to other requirements which are not yet sufficiently met in the current state of the system.

3.3.3 Prototyping in the Design Phase

Non-functional requirements refer to the realization of the system in a particular environment. Depending on the type of requirement, the effects of design decisions can either be judged by inspection or measurement of the design model or by running a prototype. Prototyping also integrates clients and users participate in the evaluation process more directly. Only limited conclusions, however, can be drawn if the prototype neither runs in the target software environment nor on the intended hardware. Conversely, the prototype cannot run completely in the target environment since this contradicts the idea of providing feedback as early as possible. This dilemma can be resolved by running a hybrid prototype which consists of portions of the original KARL prototype (i.e. the executable model of expertise) in conjunction with portions that have already been operationalized in the target language and which substitute their counterparts in the model of expertise. Specifically, those parts which are primarily affected by the design decisions to be evaluated are operationalized in the target language. Running the design prototype then comprises appropriately switching between the KARL interpreter and the interpreter of the target language (including suitable data conversions). Clearly, this form of prototyping requires the target language to be known, which is one of the reasons to restrict ourselves presently to C++ as target software environment.

Evaluation of design decisions also involves checking their admissibility with respect to functional requirements, i.e. the system must still exhibit the required functionality. In some cases, admissibility may be ensured by formal means, e.g., by employing only semantics-preserving transformations. Usually, however, admissibility is checked via testing, which in MIKE can be accomplished by comparing the results of the design prototype and the KARL prototype if both are supplied the same input data. Test cases that have already been used for evaluating the KARL prototype may be reused for checking the design prototype.

3.3.4 History and Rationale

Several authors (cf., e.g., (Potts and Bruns, 1988; Lee, 1991; Baxter, 1992)) noticed that it is often insufficient to use a description of the current version of the artefact as the only basis for further development and maintenance. In that case, it is hard to figure out which design decisions and activities have already taken place and for what reasons have been made. Thus, “a maintainer may repeat mistakes that were made by the original designer but not documented or may undo earlier decisions that are not manifest in the code” (Potts and Bruns, 1988, p. 418]. Due to the tentative nature of many design decisions, it may also be necessary to revise earlier decisions and try other design alternatives.

Therefore, the design history and the rationale behind decisions should be made explicit. The record of design activities should not just contain those steps that led to the current version of the artefact, but should also reflect revised decisions and the reasons why they failed. Non-functional requirements often interact, i.e. a decision which contributes to the satisfaction of one non-functional requirement may often inversely affect another requirement (e.g. the trade-off between storage space and processing time). Usually, design decisions interact. Introducing a particular data structure for explainability reason precludes, e.g., the removal of this data structure in order to save memory space. An account of design process and rationale must also address these relationships.

In order to address these issues, the design model in MIKE does not just consist of a description of the design *product*, i.e. the artefact under development, but also includes an account of the design *process*.

4 THE FORMALISM FOR THE DESIGN MODEL

The initial input for the design phase in MIKE is a KARL specification of the system's reasoning behaviour and of the required knowledge in the model of expertise. Design incrementally adds details which concern the realization of that behaviour and affect data structures or algorithms and the structure of the system. Thus, *DesignKARL* as the formalism for the description of the design model is an extension of KARL by language primitives addressing these design-specific aspects.

DesignKARL must also be capable of describing the design process. A basic concept in that context is the application of a particular design activity. DesignKARL must be able to express what type of activity is performed. A design activity implies a change of a particular portion of the artefact. Yet, the design activity does not create a completely new artefact, but only a new *version* of the artefact. In order to be able to refer to different versions of the design product or portions thereof, a so-called tense-shift operator @ is introduced which establishes a connection between items that are meant to be contemporary. Informally, a reference to $item_k@t_n$ denotes the specific version of $item_k$ which exists at "time" t_n. Versions can be ordered by means of a partial precedence order. Details on the operator @ can be found in (Rönnquist, 1992).

Since the rationale of design decisions must also be recorded, DesignKARL provides additional language primitives for non-functional requirements and their relationships, but also for linking requirements and their decomposition to design activities.

Due to the close relationship between non-functional requirements and the application and the effects of design decisions, a common formalism for the description of these aspects seems to be appropriate. A largely uniform description formalism across different phases of the life-cycle has the advantage that a system designer does not have to get acquainted to many different languages. As kbs design can itself be viewed as an expert task, the use of an extension of KARL for describing design product as well as design process and rationale is manifest since KARL was developed to specify knowledge-intensive tasks. In the following sections, some of the language elements of DesignKARL will be presented.

4.1 Language Elements of DesignKARL

4.1.1 Description of the Design Product

4.1.1.1 Primitives of KARL

The model of expertise in MIKE, and consequently KARL as the formalism for its description, is influenced by core ideas of KADS (Wielinga, Schreiber, and Breuker, 1992; Schreiber, Wielinga, and Breuker, 1993). Different knowledge types are separated by means of a distinct layer for each type, namely domain layer, inference layer, and task layer.

The domain layer contains the required domain-specific knowledge for the task to be solved. In KARL, domain knowledge is modelled in terms of entities and relationships. *Objects* denote entities in the domain of discourse. Features of entities are described by attributes which are encapsulated within the object, while relationships between entities are specified as *predicates*. Concepts, i.e. collections of entities exhibiting the same features, are described by *classes* which may in turn be characterized by attributes. Classes are arranged in a generalization hierarchy. Any object must be an element of at least one class and inherits all attributes which are defined for the extension of these classes. Attributes are either functional or relational. Individual objects and tuples of predicates are described by *logical expressions* which are in essence expressions of first-order logic plus some extensions in order to master object-oriented data modelling. Expressions describe which entities actually exist in the application, how they look like, and which relationships hold between concrete entities. At the domain layer, there is no account of how and when this knowledge will be used for the solution of the problem.

The inference layer specifies inference steps by means of *inference actions* and *roles*. Potential processing steps are described by means of elementary or composed inference actions. Elementary inference actions are associated with logical expressions which declaratively specify how output data are computed from input items. Composed inference actions obtain their semantics by decomposition to more basic inference actions. Roles constitute generic descriptions of inputs and outputs of inference actions and are associated with classes and predicates. There are three types of roles: *stores*, *views*, and *terminators*. Stores connect inference actions and act as containers for intermediate data. Views and terminators establish connections to the domain layer. Upward (or, respectively, downward) mappings indicate which elements of classes or tuples of predicates at the domain layer correspond to elements of generic classes or predicates associated with the views (terminators). Views read from the domain layer while terminators write to the domain layer.

The inference layer does not indicate when and in which sequence inference actions will be executed. Control of execution is specified at the task layer. To that end, complex *programs* may be built from primitive programs, i.e. calls of inference actions or assignments to boolean variables, using the usual control constructs sequence, alternative, and iteration. Programs may be combined to *subtasks*, thus establishing a counterpart to composed inference actions at the inference layer. Subtasks may in turn be used as constituents of programs.

The connection of the different layers is established by the fact that views and terminators at the inference layer access domain knowledge and calls to programs and subtasks at the task layer cause the corresponding inference actions to be executed at the inference layer. Detailed descriptions of KARL may be found in, e.g., (Angele, Fensel, and Studer, 1994; Fensel, 1993).

4.1.1.2 Extensions in DesignKARL

Important tasks in system design are describing how data and processing can be realized appropriately in the target environment on the one hand, and imposing structure on the overall system and further decomposing the resulting constituents on the other hand. Extensions introduced in addition to KARL primitives address these issues. Some of these extensions are illustrated in the context of a KARL model of the Sisyphus office assignment task (Fensel, 1993) which is used as a running example in the following sections.

4.1.1.2.1 Data Types

An object class in KARL abstracts from information which is not important from the conceptual point of view. Often, such additional information, e.g., an implicit ordering of elements of a class, can be exploited for efficient use or can facilitate the mapping to language primitives of the target environment. Therefore, in DesignKARL, a set of common *data types* is provided for making such additional information explicit. Each data type is associated with pre-defined operations that may be performed on instances of the respective type for testing or data retrieval and storage. The collection of data types comprises the types *value*, *class*, *predicate*, and *set* (which are already available in KARL) as well as the types *sequence*, *stack*, *queue*, *n-ary tree*, *hash table*, *index structure*, and *reference*. Only a restricted set of data types is offered in DesignKARL since such a restriction facilitates support for their implementation by mapping them to appropriate code fragments for particular target environments. The data types in DesignKARL are similar to those in common programming languages like, e.g., C++ or Modula, plus hash tables and index structures, which are widely used in database applications. Therefore, the collection of data types should be sufficient for most applications. Yet, new data types might be added in future versions of DesignKARL if there is a strong need to do so.

In the design phase, additional features may be used in conjunction with classes and predicates. For instance, methods, i.e. user-defined operations, may be associated with classes. Methods may apply to the class (class methods) or to its elements (member methods) and can be specified by logical expressions or by algorithms (see below). Subclasses and elements of a class inherit the methods defined in that class. During design, *abstract classes* (cf., e.g., (Rumbaugh, Blaha, Premerlani, Eddy, and Lorensen, 1991)) may be introduced as 'artificial' superclasses in order to exploit method inheritance without being required for conceptual reasons. Member methods may, e.g., be used for computing the values of derived attributes of an object. Class methods are particularly useful in views and terminators as realization of mapping expressions which connect classes at the inference layer with data items at the domain layer. In contrast to common object-oriented languages, methods may also be associated to predicates in DesignKARL since a connection between domain and inference layer can also be established via predicates associated with views and terminators.

References indicate where a particular object or data structure can be found rather than which particular features the object or data structure itself does carry. The use of *references* may be convenient if direct manipulation of data items would, e.g., involve a lot of copying or repeated search in a large collection of data items for an item with a distinctive feature. The data types *sequence*, *stack*, *queues*, and *n-ary tree* embody particular ordering relationships among the data items to be represented in the usual way. *Hash tables* and *index structures* are particularly suited for efficient access to large quantities of data. *Hash tables* are a generalization of sequences with data items being retrieved using a hash function (cf., e.g., (Cormen, Leiserson, and Rivest, 1990). The hash function has to be defined when declaring a hash table and uses a particular feature of the data items as an argument. Similarly, *index structures* (cf., e.g., (Elmasri and Navathe, 1989)) exploit a direct association between values of a particular feature of the data items and a location within the data structure used for storing them. In contrast to hash tables, this association does not involve the computation of a function to obtain potential locations of the required data item. Rather, a particular feature such as, e.g., an attribute of an object class is used as an index to the stored data.

Example 1

The use of appropriate data structures is illustrated in the context of the Sisyphus office assignment task (Linster, 1992). A solution of that problem consists of a consistent assignment of all employees to office rooms. To that end, employees are selected for being assigned in a certain order which in essence reflects the idea that employees which are most constrained with respect to possible places should be treated first. In a KARL specification of the Sisyphus problem (Fensel, 1993), this information is recorded in the binary predicate *before_comp* underlying the view *Next_Comp*. In this view, employees are modelled as elements of the (domain-specific) class *employees* and the (generic) class *components*. Attributes and attribute values of an element of a class appear in square brackets where needed. The clauses below express that, e.g., the head of the group should be assigned before any other employee, secretaries should be treated before all others except for the head of the group, etc.

VIEW $Next\_Comp@t_n$
  DEFINITIONS
    PREDICATE *before_comp*
      high : {*components*};
      less : {*components*};
    END;
  UPWARD MAPPING
    $\forall x_C, \forall y_C$
      $(before\_comp(high : x_C, less : y_C) \leftarrow x_C \in big\_boss \wedge y_C \in employees \wedge \neg (x_C \cong y_C))$.
    $\forall x_C, \forall y_C, \forall y_V$
      $(before\_comp(high : x_C, less : y_C) \leftarrow$
        $x_C[role : \text{“Secretary”}] \in employees \wedge y_C[role : y_V] \in employees \wedge$
        $\neg (y_V \cong \text{“Secretary”}) \wedge \neg (y_C \in big\_boss))$.
    $\forall x_C, \forall y_C, \forall y_V$
      $(before\_comp(high : x_C, less : y_C) \leftarrow$
        $x_C[role : \text{“Manager”}] \in employees \wedge y_C[role : y_V] \in employees \wedge$
        $\neg (y_V \cong \text{“Secretary”}) \wedge \neg (y_V \cong \text{“Manager”}) \wedge \neg (y_C \in big\_boss))$.
    $\forall x_C, \forall y_C, \forall y_V$
      $(before\_comp(high : x_C, less : y_C) \leftarrow$
        $y_C[role : y_V] \in employees \wedge x_C \in boss \wedge$
        $\neg (y_C \in boss) \wedge \neg (y_V \cong \text{“Secretary”}) \wedge \neg (y_V \cong \text{“Manager”}))$.
    ...
END;

The computation of the extension of the predicate *before_comp* may be expensive and is repeated each time an employee is placed although the order in which people are considered is static. Therefore, efficiency can be improved by introducing an additional data structure to store the ordering information explicitly. Then, the ordering can be computed once and is afterwards retrieved from the data structure rather than being re-computed. A sequence of sets is chosen as suitable data structure. Each element in the sequence consists of the set of employees which are ranked equally with respect to the ordering[1].

VIEW $Next\_Comp@t_{n+1}$
  DEFINITIONS
    PREDICATE *before_comp*
      high : {*components*};
      less : {*components*};
    END;

    SEQUENCE *ordered_components* OF SET OF *components*;

---

1. The relationship between the representations in $Next\_Comp@t_n$ and $Next\_Comp@t_{n+1}$ is specified in the description of the design decision relating the two versions of *Next_Comp*, see Example 4 at page 17.

UPWARD MAPPING
$\forall x_C, \forall y_C$
(before_comp(high : x_C*, less* : y_C*)* $\leftarrow x_C \,\varepsilon\,$ *big_boss* $\wedge\, y_C \,\varepsilon\,$ *employees* $\wedge \neg (x_C \cong y_C)$ *).*
... // ... as before

END; ♦

4.1.1.2.2 Algorithms

Besides defining appropriate data structures, an important goal of the design phase is the development of suitable *algorithms*. Knowledge acquisition focuses on describing which processing steps have to be carried out in principle in order to solve a problem. This description may have to be refined and, possibly, restructured in order to be realized efficiently. Two basic variants to describe algorithms exist in DesignKARL.

One possible form of algorithm description is very similar to the specification of control flow in KARL. An algorithm is built from primitive expressions which either assign a value to data structures or variables or invoke an inference action (or another algorithm) and assign its return values to a respective role variables. In the design phase, variables used in an algorithm may refer to any of the available data types. Primitive expressions may be combined using the control constructs sequence, iteration and alternative.

In addition, enriched inference actions are used as a preliminary form of algorithm. In that case, the description is accomplished by a set of logical clauses. As a difference to the description of inference actions during knowledge acquisition, clauses in enriched inference actions may refer to any of the available data structures by means of pre-defined operations. This type of description is intended primarily for the refinement of bodies of inference actions as a consequence of introducing data structures.

Thus, algorithms in DesignKARL are not necessarily sophisticated algorithms from, e.g., a textbook but typically result from gradual refinement of inference actions by introducing control knowledge which improves efficiency, but which is not required from a conceptual point of view. Adding control knowledge, however, should preserve the validity of the algorithm. In this context, validity means that the algorithm description yields "similar" results as the original declarative description. Yet, for validity it is not always required to reproduce the original output values exactly. If only a subset of the original output data is actually used for further processing, it is sufficient for validity if the algorithm computes just this subset. This condition, however, is not required to hold at each level of decomposition of the two corresponding representations.

Recall that modifications resulting in a different problem-solving method should not be design decisions, but should be accomplished by a new iteration of the knowledge acquisition phase since that type of modification also affects the conceptual level and not only the realisation of the system.

Example 2

In Example 1 the view *Next_Comp* is refined. In order to exploit this refinement for improving efficiency, the use of the view in inference actions has to be adapted correspondingly. The predicate *before_comp* is used in, e.g., the inference action *Create*, which is responsible for selecting employees that are to be placed next. The rules below state that an employee can be placed at all (i.e., is an element of the class *possible_comp*) if she is not already placed (*assigned_components*).

Then, an employee is to be placed next (i.e., is an element of the class *next_comp*) if no other employee is to be treated first (i.e., the employee in question is not a less preferable one):

ELEMENTARY INFERENCE ACTION $Create@t_n$

...
RULES
$\forall x_C, \forall x_S$
$(x_C[states :: \{x_S\}] \in possible\_comp \leftarrow$
$x_S \in states \wedge x_C \in components \wedge \neg\, assigned\_components(c : x_C, st : x_S)).$
$\forall x_A, \forall x_C, \forall x_S\, (assigned\_components(c : x_C, st : x_S) \leftarrow x_S[assign :: \{x_A[c : x_C]\}] \in states).$

$\forall x_C, \forall x_S$
$(x_C \in next\_comp[states :: \{x_S\}] \leftarrow$
$x_C[states :: \{x_S\}] \in possible\_comp \wedge \neg\, unpreferred\_comp(c : x_C, st : x_S)).$
$\forall x_C, \forall x_S, \forall y_C$
$(unpreferred\_comp(c : x_C, st : x_S) \leftarrow$
$x_C[states :: \{x_S\}] \in possible\_comp \wedge y_C[states :: \{x_S\}] \in possible\_comp \wedge$
$before\_comp(high : y_C, less: x_C)).$
END.

Using the refinement of the view *Next_Comp*, a new version of *Create* can achieve the same result more efficiently by using the first element of the newly introduced sequence as the set of next employees to be placed, provided the sequence is suitably adapted each time its first element is processed. As this involves additional control knowledge, the original elementary inference action is refined into a processing module (see section 4.1.1.2.3) comprising the counterpart of the original inference action plus another (very simple) algorithm which manipulates the data structure (using pre-defined operations such as *head* and *tail*).

P_MODULE $Create@t_{n+1}$
INTERFACE
DECOMPOSITION OF *Assign*;
PREMISES *States*, *Next_Comp*, *Next_Slot*s;
CONCLUSIONS *Successors, Next_Comp*;

CONTROL
(STORES: Successors) := Create_1(STORES: States; VIEWS: Next_Comp, Next_Slots);
(VIEWS: Next_Comp) := Create_2(VIEWS: Next_Comp);

INFERENCE
ENRICHED INFERENCE ACTION *Create_1*

...
RULES
$\forall x_C, \forall x_S\, (\boldsymbol{x_C \in next\_comp[states :: \{x_S\}] \leftarrow x_C \in head(ordered\_components) \wedge x_S \in states)}$.
END.

ALGORITHM *Create_2*
...
RULES
ordered_components := tail(ordered_components) .
END.
END. ♦

4.1.1.2.3 Modules

Modules allow the developer to reduce the overall complexity of the design process by decomposing the description into more manageable pieces. Modules convey the additional advantage of information hiding by making internals inaccessible to the outside. Access is only granted to data items and functionality which the module explicitly exports to the outside. As a consequence of the aim for structure-preserving design, the three knowledge layers of the model of expertise should be preserved in the design model. In order to retain the distinction of the different knowledge types, *domain modules* and *processing modules* are provided as structuring primitives.

Domain Modules. Domain modules are a means to group related domain knowledge and restrict communication with other domain and processing modules since each module is associated with an interface part to denote the classes and predicates which it imports from outside or exports to the outside. Access from outside is only possible to classes and predicates mentioned in the export part of the module interface. In the body of a module, auxiliary classes and predicates may be defined for internal use. In the rule section of the body, extensions of imported or locally defined classes and predicates are described by means of logical expressions.

Processing Modules. Processing modules allow the developer to decompose the inference and task layers of the model of expertise into smaller pieces. Due to the close relation of inference and task layer, each processing module corresponds to the decomposition of a composed inference action together with its associated subtask. The inference part of a processing module describes the roles and inference actions (or algorithms) which constitute the decomposition of the composed inference action, while the control part specifies the control flow between those inference actions (or algorithms). *Create*@t_{n+1} in the previous example constitutes such a processing module.

Like domain modules, processing modules communicate with other modules via an interface which relates a processing module to its abstraction and its decompositions according to the decomposition relationship between inference actions. The interface also signifies which data or control information the processing module exchanges with other parts of the system or external agents such as the user, but also which data are exchanged within the reasoning component, i.e. it is indicated which roles serve as input or output roles for the inference action described in the processing module and which domain modules supply domain knowledge to views and receive knowledge through terminators.

4.1.1.2.4 Components

C*omponents* are the top-level building blocks of the complete system. Basically, components and their relationships describe the architecture of the system. Each component corresponds to a functional unit of the system such as, e.g., reasoning component, user interface, explanation facility, data management unit, etc. While designing the reasoning component (which comprises reasoning mechanisms and knowledge base), other components are treated as black boxes, i.e. their internals are not detailed with DesignKARL. Only their interfaces are specified with DesignKARL. The information in the interface description of a component largely corresponds to the type of information captured in the Task and the Communication Models (de Hoog, Martil, Wielinga, Taylor, Bright, and van de Velde, 1992), or, formerly, the Model of Cooperation (de Greef and Breuker, 1992) of the KADS model suite. The interface description covers such aspects as providing references to the data exchanged, sender and recipient, and an indication who is the active participant in the transfer or what triggers the transfer.

Example 3

The reasoning component of the Sisyphus application consists of a top-level processing module *Assign* and a single domain module (which constitutes the entire domain layer). The reasoning component receives *employees* and *places* to be assigned from the user interface and returns the set of *employees* (including an attribute describing the placement) to the user interface.

```
COMPONENT Reasoning
   INPUT
      CLASS employee FROM COMPONENT Interface TO D_MODULE Domain INIT Interface;
      CLASS places FROM COMPONENT Interface TO D_MODULE Domain INIT Interface;
   OUTPUT
      CLASS employee FROM D_MODULE Domain TO COMPONENT Interface INIT Reasoning;
   P_MODULES Assign;
   D_MODULES Domain;
END.
```

♦

4.1.2 Description of the Design Process

Besides describing the shape of the system at the current stage of development, a record of how the developer arrived at that state is required since this may convey valuable information for further development and maintenance. Basically, a design decision is equivalent to a relation between two versions of parts of the system description. These relations are not part of the design product, but constitute meta information about parts of the design product. Four basic types of design decisions have been identified: *realise*, *structure*, *introduce*, and *abandon*.

Realise. During design, classes and predicates are refined to data types on the one hand and inference actions and subtasks are refined to algorithms and, possibly, processing modules on the other hand. These transitions are described by *realisations* which indicate how one version of such an item relates to its precursor. In some sense, realisations express invariant properties of the relationship between the two versions involved.

Data realisations specify a mapping between data structures similarly to the mapping between domain-specific and generic classes and predicates in views and terminators. Mappings are described as logical clauses which may include pre-defined operations on data types. These expressions do not only explicate the derivation of an arbitrary version from its precursor in order to make the design process more transparent, but are also used in the automatic conversion between representations during prototyping (see section 3.3.3). Data conversions are required if parts of the original KARL specification are coupled with parts that are already operationalized in the target language. The link between such parts is established by roles. Since data may be exchanged in both directions, the transition from abstract to refined versions has to be described as well as the inverse.

Processing realisations indicate how results computed by a task expression (which may be a complete subtask, but may also be one or more calls to inference actions arranged in a certain control sequence) relate to results computed by corresponding algorithms. Admissible relationships in a processing realisation are set equality, set inclusion, or set membership between results computed by the algorithm and results of the task expression.

Example 4

In example 1, a new version t_{n+1} of the view *Next_Comp* is created by introducing an additional data structure to explicitly represent the ordering of employees. The realisation below specifies the mapping between the newly introduced representation of the ordering by means of the sequence *ordered_components* and the previous representation by means of the predicate *before_comp*. Given an employee x_C which is ranked higher than an employee y_C by *before_comp*, both employees must be contained in the sequence *ordered_components*, and vice versa. In particular, x_C will be at a position i which is smaller than y_C's index j.

REALISE VIEW $Next\_Comp@t_n$ BY VIEW $Next\_Comp@t_{n+1}$
$\forall x_C, \forall y_C$
$(\ before\_comp(high: x_C,\ less: y_C)@t_n \leftrightarrow$
$(\exists i, \exists j\ (x_C\ \varepsilon\ ordered\_components[i] \wedge y_C\ \varepsilon\ ordered\_components[j] \wedge (0 < i) \wedge (i < j))@t_{n+1}\ ).$
END. ♦

Structure. During design, the structure of some constituents of the artefact may need to be refined or modified. For instance, portions of the model of expertise may be concerned with the management of particular data, while in the design phase it is decided to employ a dedicated data management component for that purpose. In that case, the structure of the artefact changes since data administration is now external to the reasoning component. A *structure* relationship connects the two versions of the affected module. A structure relationship is also used if the interfaces of modules are modified or an additional level of decomposition is introduced, e.g., by refining an elementary inference action into a processing module as a consequence of additional control information.

Recall that design should aim at preserving the structure of the model of expertise. Therefore, structural changes beyond refinement of parts of the artefact should only be made if there are good reasons for doing so. In addition, it should be made sure that structural changes leave the conceptual level intact. Otherwise, the modifications should be carried out in a new cycle through the analysis phase.

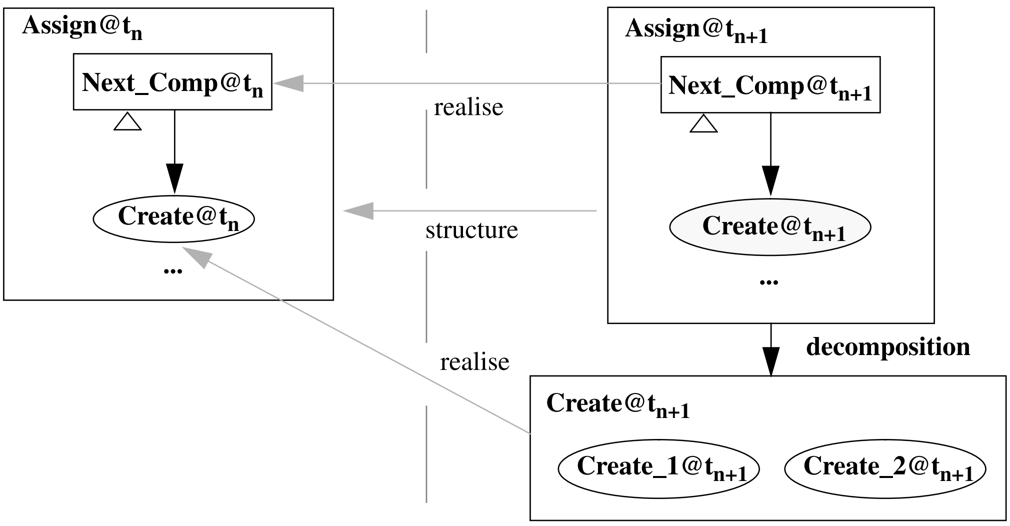

Figure 2 Schematic depiction of design decisions

Example 5

In example 2, the elementary inference action *Create* is refined into a processing module. As a consequence, the structure of the processing module *Assign* which contains *Create* as part of its decomposition has to be adapted since an additional level of decomposition is introduced:

STRUCTURE P_MODULE $Assign@t_n$ INTO P_MODULE $Assign@t_{n+1}$
DECOMPOSITION OF INFERENCE ACTION $Create@t_n$

Figure 2 summarizes the design decisions that have been made in examples 1 to 5. Large boxes indicate processing modules, while the small boxes denote roles (in particular, views). Circles indicate inference actions or algorithms. ♦

Introduce and Abandon. Design decisions may *introduce* constituents which do not have a counterpart in earlier versions of the artefact. For instance, functions for handling communication with other components of the system may be introduced as well as error handling routines for reasonable reactions to unexpected situations in order to increase system reliability. These issues usually do not show up in the model of expertise since they do not affect the solution of the overall task.

Conversely, portions of the artefact may be removed, e.g. because the corresponding issue is decided upon not be realised due to an incompatibility with some non-functional requirement. In that case, the last appearance of the affected constituents are marked with an *abandon* relationship.

4.1.3 Description of Design Rationale

Design decisions are motivated by certain requirements that have to be met in the final implementation. In order to make the development process more transparent and thus support further development and maintenance, the rationale behind particular design decisions should be made explicit. The model employed in MIKE for the description of design rationale and its connection to non-functional requirements is based on ideas developed in (Potts and Bruns, 1988; Lee, 1991) and (Chung, 1991; Mylopoulos, Chung, and Nixon, 1992). For the sake of brevity, however, the description of design rationale will not be detailed here, but will be subject of a future paper.

5 RELATED WORK

Work on the design phase in KADS (van de Velde, Duursma, and Schreiber, 1993; Schreiber, 1993; Terpstra, 1993) is closely related to the work reported here. The scope of the KADS design model is broader than the one in MIKE since KADS commits itself neither to particular analysis or design paradigms (such as function-oriented or object-oriented), nor to specific classes of implementation environments. Therefore, on the one hand, an explicit selection of a design paradigm and a decomposition according to that paradigm have to be carried out in the design phase and, on the other hand, the mapping to the target environment may be more difficult since design paradigm and implementation environment may differ considerably in spirit. These issues are presently less important in MIKE as a consequence of the restrictions we made at this stage of the project.

Like MIKE, KADS also adopted the idea of a risk-driven iterative life-cycle. In particular, building a model is viewed as a process of configuring, refining, and assessing a generic template for the respective model. This point of view is comparable to the one taken in MIKE, except that MIKE does not emphasize the idea of model templates.

KADS does not prescribe a particular formalism for the model of expertise. The range of description formalisms may include semi-formal notations as well as formal languages. Therefore, the description of the design model cannot make assumptions about the shape of the model of expertise and, thus, it is not possible to describe design decisions as detailed as in MIKE.

KADS views the task of making the model of expertise operational as part of the design phase (Schreiber, 1993). Thus, extra effort is required if the model of expertise shall be evaluated by means of a prototype. In contrast, MIKE views prototyping as an integral part of the knowledge acquisition phase. Consequently, the construction of an executable description of the model of expertise cannot be deferred to the design phase, but has to be part of the knowledge acquisition phase. As a consequence, the description of the model of expertise has to be much more detailed than in KADS.

The design phase in MIKE is to some extent influenced by "conventional" design approaches. The basic issues addressed during design in MIKE, namely defining an appropriate system structure, decomposing subsystems into smaller units such as modules, and developing appropriate data structures and algorithms, can be found quite similarly in structured design approaches (e.g. (Yourdon and Constantine, 1978)) as well as object-oriented design methods such as (Coad and Yourdon, 1991; Rumbaugh, Blaha, Premerlani, Eddy, and Lorensen, 1991). The design approach in MIKE tries to combine benefits of both paradigms: real world entities are modelled as objects and object classes, which is a quite intuitive representation, but processing steps are represented on a global level instead of distributing responsibility for such steps to different object classes. Thus, the problem-solving method as it is expressed in the model of expertise can be identified more easily in the final implementation, which may prove valuable for system maintenance and which conforms to the principle of structure-preservation in a better way than a purely object-oriented approach. Design in MIKE pays particular attention to the non-functional requirements which constitute the rationale for design decisions. This emphasis on specifically non-functional requirements is not normally part of design approaches in software engineering.

6 CONCLUSION

An *executable* description of the model of expertise with KARL constitutes the basis for the design phase in MIKE. Consequently, functional requirements towards the system have already been described and evaluated in the model of expertise, and therefore, the design phase is driven by non-functional requirements. Design decisions basically affect the *structure of the system* and *the algorithms and data structures* employed and are made in an incremental fashion. Due to the tentative nature of design decisions, there is a need for early evaluation of design decisions by means of a prototype which is derived from the KARL prototype used during knowledge acquisition by substituting portions of it by corresponding parts operationalized in the target language.

In order to make the design process more transparent, we argue that it is insufficient to describe only the shape of the artefact in the design model. Instead, we propose to also document the history of design decisions and the motivation behind them by relating them to non-functional requirements. DesignKARL as an extension of KARL is used as the formalism for the description of the design product as well as the design process and rationale within the design model. A description of the design model which also addresses design process and rational may serve as a thorough documentation of the design and, thus, may support maintenance. Likewise, such a documentation may support the reuse of design specifications since it is much easier to decide which design decisions still apply in a different context (e.g., different task, different target environment, etc.). Clearly, the documentation of design process and rationale poses an additional workload on the designer. Yet, the designer may be relieved from other low level tasks since this information may be exploited for intelligent support of the design process, e.g., by supporting the selection of design decisions and keeping track of dependencies, but also by automating parts of the development through the provision of automated transformations of standard data types into code fragments of the target environment. In order to strike a balance between additional overhead and benefit gained, we adopted a model for design rationale which does not contain some of the aspects included in the proposals of (Potts and Bruns, 1988; Lee, 1991) such as, e.g., a detailed description of the argumentation for or against design alternatives.

The considerations we made are less important if a kbs is constructed on the basis of an already available shell since in that case, most design decisions have alredy been made while the shell was built and cannot be influenced when building an application. Thus, there is only limited need for a distinct design phase. Shell-based approaches, however, have other disadvantages: general-pupose shells do not provide a conceptual model to guide knowledge acquisition while role-limiting shells incorporate a strong model of the problem-solving method, but usually lack flexibility when the built-in problem-solving method does not exactly fit the problem at hand.

Clearly, tool support is required in order to make the design framework of MIKE easier to use. Work on such a tool environment is in progress. This includes the extension of the KARL interpreter, but also some types of automated support, such as generation of code fragments for the standard data types in DesignKARL as well as data conversions between implemented and specified parts of the design product. Furthermore, the documentation of the design process and rationale will be supported by the tool environment. Current work also includes the evaluation of the proposed design framework by using it on applications of realistic size such as, e.g., the Sisyphus elevator configuration task (Poeck, Fensel, Landes, and Angele, 1994).

Acknowledgement

We gratefully acknowledge the discussions with Jürgen Angele and Dieter Fensel and their comments on drafts of the paper.

References

Angele, J. (1993). *Operationalisierung des Modells der Expertise mit KARL.* DISKI 53, infix-Verlag, St. Augustin, Germany (in german).

Angele, J., and Fensel, D. (1992). A spiral model for knowledge engineering. Research report 245, Institut für Angewandte Informatik und Formale Beschreibungsverfahren, University of Karlsruhe.

Angele, J., Fensel, D., Landes, D., Neubert, S., and Studer, R. (1993). Model-Based and Incremental Knowledge Engineering: The MIKE Approach. In *Knowledge Oriented Software Design*, J. Cuena (Ed.). IFIP Transactions A-27, Elsevier, Amsterdam, 139-168.

Angele, J., Fensel, D., and Studer, R. (1994). The model of expertise in KARL. In *Proceedings of the 2nd World Congress on Expert Systems* (Lisbon/Estoril, Portugal, January 10-14).

Baxter, I.D. (1992). Design maintenance systems. In *Communications of the ACM 35(4)*, 73-89.

Boehm, B.W. (1988). A spiral model of software development and enhancement. In *IEEE Computer 21*, 61-72.

Chung, L. (1991). Representation and utilization of non-functional requirements for information system design. In *Ad-*

vanced Information Systems Engineering, R. Andersen et al. (Eds.). LNCS 498, Springer, Berlin, 1991, 5-30.

Cormen, T.H., Leiserson, C.E., and Rivest, R.L. (1990). *Introduction to Algorithms*. MIT Press, Cambridge.

Coad, P., and Yourdon, E. (1991). *Object-Oriented Design*. Yourdon Press, Englewood Cliffs.

Elmasri, R., and Navathe, S.B. (1989). *Fundamentals of Database Systems*. Benjamin/Cummings, Redwood City.

Fensel, D. (1993). The knowledge acquisition and representation language KARL. Doctoral dissertation, University of Karlsruhe, Germany.

Floyd, C. (1984). A systematic look at prototyping. In *Approaches to Prototyping*, R. Budde et al., eds. Springer, Berlin, 1-18.

de Greef, P., and Breuker, J. (1992). Analysing system-user cooperation in KADS. In *Knowledge Acquisition 4(1)*, 89-108.

Guida, G., and Mauri, P. (1993). Evaluating performance and quality of knowledge-based systems: foundation and methodology. In *IEEE Transactions on Knowledge and Data Engineering 5(2)*, 204-224.

de Hoog, R., Martil, R., Wielinga, B., Taylor, R., Bright, C., and van de Velde, W. (1992). The CommonKADS model set. Technical report KADSII/T1.2/WP1-2/TR/UvA/0040/4.0, University of Amsterdam.

Keller, S.E., Kahn, L.G., and Panara, R.B. (1990). Specifying software requirements with metrics. In *System and Software Requirements Engineering*, R.H. Thayer and M. Dorfman (Eds.). IEEE Computer Science Press, Los Alamitos, 145-163.

Landes, D. (1993). Development of knowledge-based systems on the basis of an executable specification. In *Expertensysteme '93*, F. Puppe and A. Günter, eds. Springer, Berlin, 139-152 (in german). English version available as research report 265, Institut für Angewandte Informatik und Formale Beschreibungsverfahren, University of Karlsruhe.

Lee, J. (1991). Extending the Potts and Bruns model for recording design rationale. In *Proceedings of the 13th International Conference on Software Engineering* (Austin, Texas, May 13-17), 114-125.

Linster, M. (Ed.) (1992). Sisyphus'92: Models of problem solving. Arbeitspapiere der GMD 630, GMD, St. Augustin, Germany.

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

Neubert, S. (1993). Model construction in MIKE. In *Knowledge Acquisition for Knowledge-Based Systems,* N. Aussenac et al. (Eds.). Lecture Notes in Artificial Intelligence 723, Springer, Berlin, 200-219.

Poeck, K., Fensel, D., Landes, D., and Angele, J. (1994). Combining KARL and configurable role-limiting methods for configuring elevator systems. In In *Proceedings of the 8th Knowledge Acquisition for Knowledge-Based Systems Workshop KAW'94* (Banff, Canada, January 30 - February 4).

Potts, C., and Bruns, G. (1988). Recording the reasons for design decisions. In *Proceedings of the 10th International Conference on Software Engineering* (Singapore, April 11-15), 418-427.

Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., and Lorensen, W. (1991). *Object-Oriented Modelling and Design*. Prentice Hall, Englewood Cliffs.

Rönnquist, R. (1992). Theory and practice of tense-bound object references. Doctoral dissertation, University of Linköping, Sweden.

Schreiber, G. (1993). Operationalizing models of expertise. In (Schreiber, Wielinga, and Breuker, 1993), 119-149.

Schreiber, G., Akkermans, H., and Wielinga, B. (1990). On problems with the knowledge level hypothesis. In *Proceedings of the 5th Knowledge Acquisition for Knowledge-Based Systems Workshop KAW'9*0 (Banff, Canada, November 4-9).

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

Terpstra, P. (1993). Personal communication.

van de Velde, W., Duursma, C., and Schreiber, G. (1993). Design model and process. Technical report KADSII/M7/VUB/RR/064/0.1, Free University of Brussels.

Wielinga, B.J., Schreiber, A.Th., and Breuker, J.A. (1992). KADS: A modelling approach to knowledge engineering. In *Knowledge Acquisition 4(1)*, 5-53.

Williams, D.O., Tomlinson, C., Bright, C.K., and Rajan, T. (1992). The CommonKADS quality viewpoint. Technical report KADSII/T2.2/TR/LR/0040/1.0, Lloyd's Register, London.

Yourdon, E., and Constantine, L. (1978). *Structured Design*. Yourdon Press, New York.