An Approach to the Design of Knowledge-Based Systems

Dieter Landes
Institut für Angewandte Informatik und Formale Beschreibungsverfahren
Universität Karlsruhe, D-76128 Karlsruhe, Germany
e-mail: landes@aifb.uni-karlsruhe.de

ABSTRACT

The issue how to achieve a principled transition from knowledge acquisition to implementation such that a knowledge-based system (kbs) of high quality results is rarely discussed in the knowledge engineering literature. Here, this topic is addressed in the context of the design phase of the MIKE approach to kbs development. In MIKE, design decisions are motivated primarily by non-functional requirements. Due to the iterative nature of the design process, design decisions have to be evaluated early. In MIKE, this can be achieved by experimental prototyping, but also by quantitative measures associated to requirements. Furthermore, in order to improve transparency and traceability, the description of the design includes a record of the design decisions and their motivation in addition to a description of the artefact. DesignKARL is sketched as the formalism for expressing these aspects.

INTRODUCTION

In recent years, rapid prototyping as the prevailing paradigm for building kbs turned out to be insufficient for building large-scale systems for long-term use (cf. e.g. [1]). In particular, the complexity of the development task is nearly unmanageable since 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, leading to poorly structured systems. As another drawback, the final implementation often constitutes the only documentation of the expertise embodied in the kbs.

More principled methods for constructing kbs are required for being able to build large systems to be used and maintained over a long period of time. MIKE (Model-based Incremental Knowledge Engineering) [2] has been proposed as such a principled framework. MIKE aims at integrating the benefits of life-cycle models, prototyping, and formal specifications. The focus of this paper lies on the design phase of the MIKE life-cycle and the activities in that phase that are intended to improve the quality of the developed system. A short account of the basic principles of MIKE will be given in order to clarify the context of the design phase. Then, the design approach in MIKE will be characterized. An important feature of our design approach is the timely evaluation of design decisions by means of prototyping, but also by rating their success on the basis of quantitative measures. Furthermore, for reasons of transparency and traceability, the design process and its underlying rationale must be recorded in addition to a description of the design product. DesignKARL [3] is sketched as the formalism for describing the design product as well as the design process and its rationale.

CORE IDEAS OF MIKE

The main goal of an expert system is the ability to adequately solve a task which normally requires expert knowledge to be carried out. In particular, the knowledge *how* to solve the task is an indispensable part of the expertise. In contrast to earlier opinions, it is nowadays widely accepted that the required knowledge cannot simply be extracted from an expert and transferred to a computer system. Rather, knowledge engineering has to be viewed as an incremental modelling process. Due to that characteristic, early feedback by reality is required and model revisions must be possible in any stage. Furthermore, in order to master the overall complexity, experience from software engineering indicates that development must be split into several steps each of which concentrates on particular aspects of the overall task while abstracting from others. In MIKE, these steps are *knowledge acquisition*, *design*, *implementation*, and *evaluation*. Consequently, different representations are provided each of which is particularly suited for one of the development steps or substeps thereof.

The first substep in the knowledge acquisition phase consists of obtaining a natural language description of the expertise. This knowledge then has to be interpreted, giving rise to a semi-formal, but structured description in the so-called *hyper model* [4]. Interpretation is followed by formalization, resulting in a so-called *model of expertise*, the shape of which is strongly influenced by ideas from KADS [5, 6]. In particular, three basic knowledge types are distinguished each of which is represented at a different layer of the model of expertise. The representation formalism for the model of expertise in MIKE is the formal specification language KARL [7, 8]. Besides its model-theoretic semantics, KARL also possesses an operational semantics [9], which makes it possible to evaluate the model by basically explorative prototyping [10]. 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 considered task can be solved, but abstract from how this solution will finally be realized on a computer. The latter issue is addressed in the design phase and described in its result, the design model. The design model then contains an account of the requirements the kbs has to fulfil and indicates how they can be met. The design model serves as the basis for implementing the requirements in the final system as the result of the implementation phase. Finally, in the evaluation phase, it has to be determined if the implemented system in its entirety actually meets the posed requirements.

Due to the iterative and approximative character of the development process, the classical waterfall model for system development is unsuitable. Rather, the life-cycle phases are traversed in a cyclic fashion similar to Boehm's spiral model [11]. Furthermore, each of the phases consists of several subphases which in turn are traversed in a cyclic manner (see Figure 1). More details on the life-cycle of MIKE can be found in [2, 9].

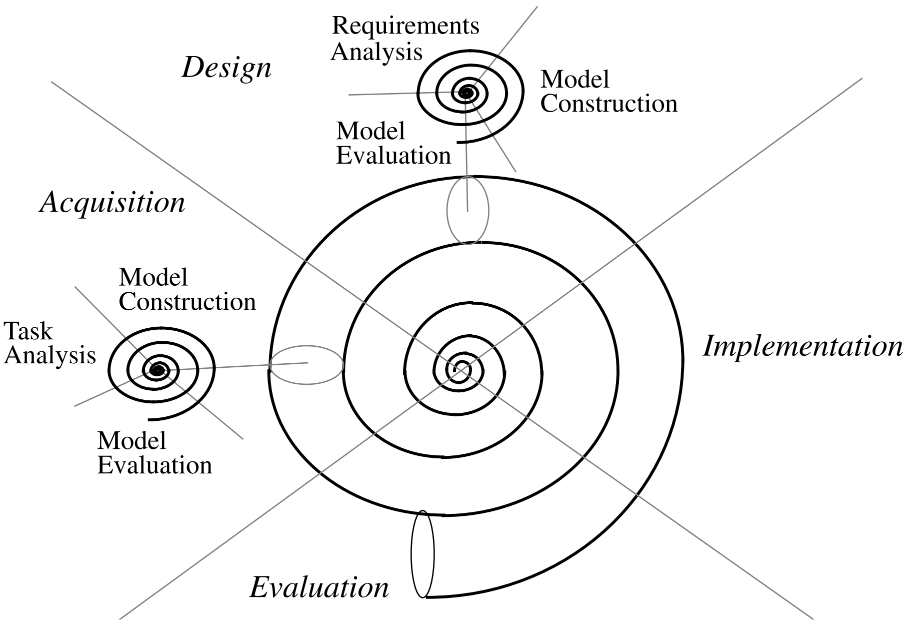

Figure 1 The knowledge engineering process in MIKE (reprinted from [2])

THE BASIS FOR THE DESIGN PHASE

The goal of the knowledge acquisition phase is the development of a model of how a computer system can solve a particular expert task. Due to the nature of the development process, timely feedback is required in order to find out if the system can really solve the task adequately. This can be achieved by evaluating a simulation of the behaviour of the future system on the basis of the developed model of expertise. To that end, MIKE uses the *executable* specification language KARL [7, 8]. As a consequence, the resulting document of the knowledge acquisition phase is a running prototype of the system's reasoning component.

Usually, a kbs also encompasses constituents such as, e.g., a user interface, a data management component, etc. besides it reasoning core. The user interface is particularly important for the success and acceptance of the kbs. At the current stage of the project, however, MIKE currently focuses on the problem-solving core of the system, i.e. reasoning mechanisms and knowledge base, since development issues concerning the other constituents are already much better understood through comparable work in "conventional" software engineering. Yet, this does not imply that the development of these external components and the problem-solving component should be carried out in complete independence. Rather, they should be developed in parallel to make sure that existing dependencies can be accounted for in due time.

Running the model of expertise allows evaluating it by means of explorative prototyping [9]. Therefore, it can be assured early during development that the specification covers the functional requirements correctly. While knowledge acquisition addresses only the functional requirements, the design phase can focus on *non-functional requirements* (cf. also [12]) since the latter requirements constrain how functionality may be realized. The difference between the types of requirements addressed in the knowledge acquisition phase and in the design phase is less clear cut in many other approaches than it is in MIKE. Normally the question how to realize some required functionality during design includes the issue of inventing an appropriate computational solution which supplies that functionality. In MIKE an operational, though mostly inefficient, solution is already developed during knowledge acquisition, and in the design phase this solution is only refined (or modified) such that it conforms to additional non-functional requirements.

So far, only some authors [13, 14] particularly focus on non-functional requirements. [15] and [16] provide taxonomies of factors that determine the quality of knowledge-based systems, but do not distinguish functional and non-functional aspects. On the basis of these taxonomies, *efficiency*, *maintainability*, *understandability*, *reliability*, *portability*, and *requirements resulting from the chosen hardware or software environment or the system architecture* have been identified as important top-level non-functional requirements for the design phase in MIKE (but the list may still require extension). It should be noted that efficiency plays a twin role since two aspects have to be distinguished: efficiency of the problem-solving method itself and efficiency of its realization [17]. The first aspect is addressed during knowledge acquisition since it affects functionality, whereas the second aspect constitutes a non-functional requirement.

DESIGN ACTIVITIES AND DesignKARL

In MIKE, the formalism for the description of the model of expertise is the specification language KARL. Inspired by KADS [5, 6], KARL distinguishes three basic types of knowledge, namely domain knowledge, inference knowledge, and task knowledge, each of which constitutes a separate knowledge layer. For all three types of knowledge, KARL provides appropriate language primitives. Domain knowledge is described primarily in terms of objects, object classes and predicates, while inference knowledge is expressed by means of inference actions and generic object classes and predicates which are associated to so-called knowledge roles, which, roughly speaking, serve as input and output parameters for inference actions. The bodies of inference actions as well as rules for describing the extension of object classes and predicates are specified declaratively by means of logical expressions. The control over the execution of inference actions is expressed declaratively using dynamic logic [18]. For legibility, these expressions are rewritten into constructs that are familiar from structured programming, such as sequence, alternative, and iteration. The connection between the different types of knowledge is established by the fact that calls to primitive programs at the task layer cause the corresponding inference actions to be executed. On execution, inference actions use the contents of their input roles, i.e. the extension of the classes and predicates which are associated to these roles, for computing the extension of classes and predicates associated with their output roles. For some roles, a connection to domain knowledge can be specified by logical expressions which indicate how domain-specific knowledge items correspond to items which are associated with the respective roles.

The language primitives available in KARL largely preclude to “program” a solution to the problem at hand already during knowledge acquisition. Rather, the knowledge engineer is forced to specify what is needed for solving the problem in principle, yet without caring for an efficient realization. Usually, efficiency constitutes the most important requirement that has to be met by the final system besides being able to solve the problem at all. Since the issue of an efficient realization constitutes a non-functional requirement, it is addressed in the design phase. To that end, DesignKARL as an extension of KARL contains additional language primitives which are targeted to support efficiency considerations. Efficiency can be built into the solution in the usual way, namely by developing appropriate algorithms and introducing suitable data structures which are employed to refine items of the declarative specification. DesignKARL offers a collection of standard data types with predefined operations that may be performed on data items of the respective types. The data types provided are value, class, predicate, set, sequence, stack, queue, n-ary tree, hash table, index structure, and reference. Algorithms can be described either in a logic-based fashion or in an "imperative" fashion. The logic-basic variant resembles the specification of bodies of inference actions since the description of an algorithm is basically made up of a set of logical clauses. However, in contrast to the specification of inference actions, more sophisticated data structures and their predefined operations may now be used. The "imperative" specification of algorithms looks similar to control flow specifications at the task layer of the model of expertise, but once again dropping the limitation of KARL to a very restricted collection of data types.

Structuring the Model

Besides refining the model of expertise due to efficiency considerations, requirements such as maintainability or reusability may require to impose a particular structure on the model, thus reducing the interconnectivity between parts of the model. Design in general is even characterized as "warfare against interconnectivity" [19, p. 260]. Furthermore, appropriate structure also reduces the overall cognitive complexity of the design, thus making the design easier and less error-prone to implement for a programmer and consequently leading to higher software quality [20]. In conventional software engineering, modules have been proposed as a means to achieve such a reduction of interconnectivity or complexity, with the additional advantage of hiding irrelevant details to parts that need not care about them (cf., e.g., [21]). In order to achieve these benefits for kbs as well, DesignKARL offers domain modules and processing modules as structuring primitives.

Domain Modules Domain modules collect related domain knowledge in a single place. Each domain module contains definitions of data structures (including classes and predicates) and rules describing the extension of these data structures. As usual, a module may use of knowledge defined elsewhere and, conversely, supply knowledge to other modules. The module interfaces restrict access to external knowledge: knowledge defined elsewhere may be used only if it is mentioned in the import section of the module that intends to use it and if another module exists that makes the knowledge publicly available in its export section. In the module body, extensions of imported or local data structures are described by means of logical expressions. A subset of these data structures may be exported to other modules. The export of a data structure implies that its extension will also become known to the importing module. The rules defining the extension are not accessible, but determine the semantics that must be respected by the importing modules.

Clusters Modularization defines a horizontal partitioning of the domain layer with respect to the degree of abstraction. Nevertheless, large domain layers may still require additional means to facilitate an understanding of the model. A similar problem is encountered in database applications if conceptual models grow so large that the entity relationship (ER) diagrams describing them become virtually unreadable. Several proposals (cf., e.g., [22, 23]) aim at alleviating this problem by means of ER model clustering, thus establishing hierarchies of ER diagrams.

Since the language primitives for describing the terminological structure of an application in KARL closely correspond to those of an extended ER diagram, clustering can be applied to KARL domain layers as well. Class clustering (corresponding to entity clustering in ER clustering approaches) allows to condense object classes into a cluster if they are related by is-a-relationships. In addition, classes that constitute the range restrictions of attributes of such object classes may be included in the cluster, which corresponds roughly to the notion of dominance grouping as proposed in [23]. However, we take a slightly more liberal stance in that relationships between the range restrictions may be included in the resulting cluster as well. In the superordinate schema, clusters resulting from this type of clustering can be used like simple object classes.

Complex predicate clustering (corresponding to complex relationship clustering in ER clustering approaches) allows to collapse a sub-diagram of the graphical notation of a KARL domain schema into a cluster which can be used like a simple predicate in superordinate diagrams [23]. In particular, this comprises the formation of a predicate cluster from a collection of semantically similar relationships.

Certain rules must be obeyed to ensure that clusters behave consistently with their refinements. These rules define how the links between a cluster and its environment, i.e. all the entity types or relationship types to which the cluster is connected, have to be set when the cluster is replaced by its refinement. These rules are expressed in DesignKARL like the transformations in [22] by indicating the set of items in the higher-order diagram that has to be replaced by another set of items in the refinement and vice versa. Basically these rules define a production rule of a graph grammar.

In contrast to modules, which prescribe a partitioning of the implementation (provided the implementation environment includes the possibility to define modules), clustering is only a means to improve understandability during the development, but which has no counterpart in the implementation. Clustering is useful early in the design process since clusters may indicate which parts of the knowledge might be candidates for being encapsulated in a module. Conversely, if modules have already been established, clusters cannot be formed across module boundaries. If the need to do so arises, this might be an indication that the module structure should be critically reviewed.

Processing Modules In KARL, it is possible to decompose inference actions and subtasks into more elementary components. *Processing modules* impose structure on the inference and task layer in addition to this type of refinement. Because of their close relationship, modules are not introduced separately at the two layers, but each processing module corresponds to the refinement 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 refinement of the composed inference action, while the control part specifies the control flow between those inference actions (or algorithms). Thus, the distinction between the different types of knowledge is still retained.

Like domain modules, processing modules communicate through interfaces. Similar to the situation at the domain layer, two dimensions of partitioning are established: modules can interact with modules at the same level of refinement, but also with modules which are part of their refinement. In contrast to the clustering mechanism at the domain layer, refinement at the inference and task layers prescribes a decomposition which usually will also be reflected in the implementation. The interface relates a processing module to its abstraction and its refinements according to the refinement relationship between inference actions. In addition, the interface 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 actions described in the processing module and which domain modules supply data to input or output roles.

During knowledge acquisition, decomposition is accomplished like the decomposition of data flow diagrams in Structured Analysis [24]. Usually, this results in a tree structure, any module being part of the refinement of at most one other module. This may cause redundancy as multiple copies of basically the same inference step may appear in the decomposition. In order to remove these redundancies, DesignKARL allows to define parameterized processing modules which may be used at various places and which are dynamically instantiated as needed.

Structure-preservation In principle, the use of the structuring primitives supplied by DesignKARL is not restricted. However, using these primitives freely makes it much harder to determine which portions of the model of expertise correspond to a particular section of the design model. [25] argues that the possibility to relate parts of the design and, later, parts of the implemented system to the sections of the model of expertise they were derived from proves to be advantageous in several respect. For instance, in case that the structure of the model of expertise is preserved during design, this relationship may be exploited for supporting the maintenance of the system. Similarly, structure-preserving design may improve the knowledge-based system´s explanation capabilities since explanations may resort to more intuitive representations in the model of expertise or even the hyper model rather than only referring to implementation-biased representations. Therefore, structural modifications which go beyond refining existing structures should be employed carefully or avoided altogether unless there are good reasons for them. Yet, restructuring cannot be avoided if the target software environment is not compatible with the structure embodied in the model of expertise.

THE DESIGN PROCESS

The design phase in MIKE begins when the model of expertise has been evaluated, i.e. when it has been assured that the reasoning component of the kbs actually fulfils the posed functional requirements. Therefore, non-functional requirements play the dominant role in the design phase. Then, design proceeds by *analysing* the requirements that have been posed in order to pick the one to be addressed next, for instance, by rating requirements by risk and selecting the one associated with the highest risk.

When a requirement has been selected, the designer looks for design activities which are likely to contribute towards fulfilling the selected requirement. As a consequence of such activities, the shape of the design model will change, giving rise to a new version of the model. Thus, performing design activities can be interpreted as a *model construction* step. The process of arriving at suitable activities can be viewed as first posing a top-level goal, namely to reach a situation in which the model of the reasoning component meets the chosen requirement. In general, such a top-level goal cannot be met all at once. Rather, the goals will be decomposed into subgoals (cf., e.g., [14, 26]). Some of these subgoals must be fulfilled jointly to satisfy the higher-level goal while others constitute alternative subgoals such that the higher-level goal is fulfilled if any of the subgoals is met, thus giving rise to an AND/OR tree of goals. Basically, two aspects may lead to a decomposition: a requirement (i.e. a goal) may either be reduced to a collection of more basic requirements or the scope of the requirement, i.e. the portion of the design product to which it refers, may be made more specific. Decomposition to more basic goals is continued until elementary goals are reached which can be achieved directly by appropriate *design decisions*, such as, e.g., refining a particular data structure to a more elaborate one. Thus, a design activity is composed by a collection of design decisions.

<u>Example</u> Let us illustrate this view of the model construction step by means of an example. In a simple scheduling task [27], the requirement for efficiency of the overall scheduling task can be met by taking care for the efficiency of the subtask which proposes new potential schedules. Proposing a new schedule involves finding and extending the most recent partial schedule. This problem can be solved efficiently either by storing only the most recent schedule, thus reducing storage needs, or by recording earlier schedules in such a way that the last one is easily accessible. The first alternative, however, is discarded since information on previous partial schedules is necessary for explaining how the system arrives at a solution. Therefore, the second option is pursued, resulting in several design decisions, such as, e.g., realizing parts of the model by appropriate data structures and abandoning earlier, less efficient ways of identifying the desired schedule. The conjunction of these design decisions achieves the desired goal. The resulting decomposition is depicted schematically in figure 2. ♦

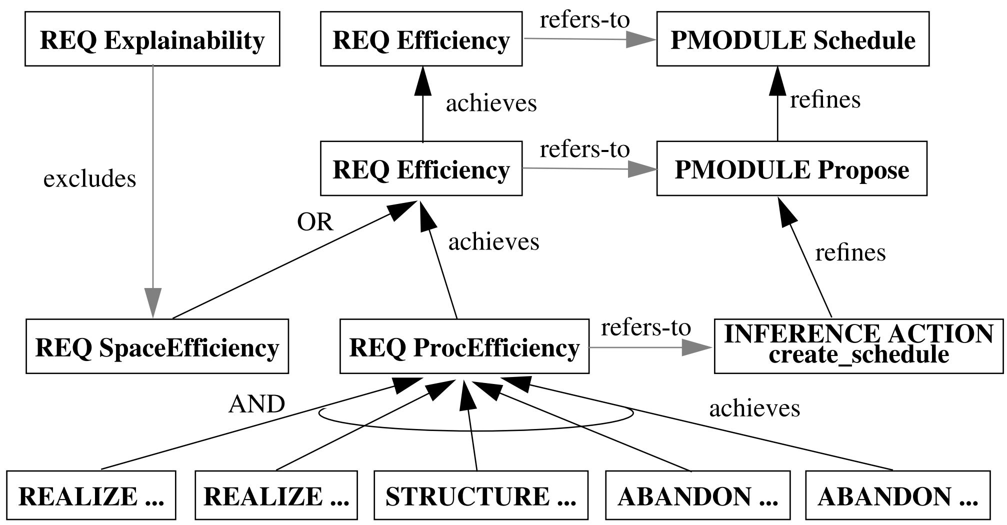

Figure 2 Schematic decomposition of requirements

After one or more design activities have been carried out, their effects have to be *evaluated*, i.e. it must be checked if they actually improved the status of the design model with respect to the selected requirements. As a second aspect of evaluation, it has to be ascertained that the system is still capable of exhibiting the required functionality as it is specified in the model of expertise. Two basic strategies are possible for the evaluation of the design model in MIKE: measurement and prototyping.

Measurement

Measurement of performance and quality is viewed by many authors as an important step in the process of turning software development from an art into an engineering discipline. This holds for "conventional" software (cf., e.g., [20]) as well as for kbs (cf., e.g., [15]). "During kbs design and construction, methods and tools are needed for measuring the degree of performance and the quality achieved by the system [...]" such that "[...] the work done is reviewed and assessed, wrong design decisions and bad construction steps are corrected, and appropriate indications for the continuation of the project are provided" [15, p. 204]. Specifically, "the greatest potential leverage for software measurement lies in design, not code, analysis" [20, p. 3]. Adopting this philosophy for the design phase in MIKE, each non-functional requirement should, whenever possible, be linked to a quantitative measure or at least a qualitative rating should be possible if this cannot be achieved. For instance, for efficiency, suitable measures may be time complexity of algorithms or storage needs of data structures employed or simply the runtime behaviour of particular parts of the model. Maintainability or understandability can be linked to complexity: Card [20] shows in a case study how complexity may be used to predict, e.g., the effort required for maintaining a software system. In his approach, the complexity of a modular software design is determined by two factors, namely the interconnectivity between modules and the internal complexity of individual modules, i.e. the average number of "decisions" in the modules.

Such measures may then be used for rating the progress made towards the fulfilment of a requirement if the desired value of the measure can be estimated on the basis of similar projects, or for choosing between design alternatives. In MIKE, no particular measures are prescribed - development of metrics is an active field of research which may supply new measures that may turn out useful for a particular kbs development effort. Individual measures may then be combined according to the framework outlined in [15].

Prototyping

Still, the definition of appropriate measures may not be viable for all of the posed requirements. But even those requirement for which no suitable measure can be supplied can be evaluated by means of prototyping. In contrast to prototyping in the knowledge acquisition phase which focuses on finding out what the functional requirements really are, prototyping in the design phase aims at the evaluation of design choices. Thus, the flavour of prototyping turns from explorative to experimental [10]. On the one hand, critical design choices have to be evaluated as soon as possible. On the other hand, only limited conclusions can be drawn if the prototype is run in an intermediate setting, i.e. neither in the target software environment, nor on the intended hardware. In order to meet both goals, the prototype in the design phase is a hybrid prototype which is composed of parts of the KARL specification, which are executed by means of the KARL interpreter, and other parts which have already been operationalized in the target language. Currently, work is in progress which aims at facilitating the operationalization for a selected target language (C++) by providing automated support in the conversion of DesignKARL primitives into primitives of the target language.

When design decisions have been made, not only the design decisions as such, but also their admissibility must be checked, i.e. it must be ascertained that the system exhibits the required functionality after all. Admissibility might also be ensured by formal means, e.g., by restricting design decisions to semantics-preserving transformations. In MIKE, a more pragmatic stance is taken by checking admissibility by means of testing. To that end, the fact that the model of expertise is still available as an operational prototype can be exploited. Thus, the results of running the DesignKARL prototype can easily be compared with those of the KARL prototype. Furthermore, test cases that have already been used in the evaluation of the KARL prototype during knowledge acquisition may now be used again for checking the design prototype.

Prototyping also offers the possibility of integrating customers and users in the evaluation process more tightly, which might lead to the clarification which non-functional requirements must be met. This is true for the reasoning component, but even more so for the user interface as the front-end of the system. Therefore, prototyping should not be limited to the reasoning component, but include other components as well. Yet, prototyping other constituents of the system is beyond the scope of this paper.

DESCRIPTION OF DESIGN PROCESS AND RATIONALE

Various authors (cf. e.g. [28, 29, 30]) argue that it is insufficient to only use a description of the artefact as it currently stands as the basis for development and maintenance. If no additional information is available, it is hard to find out which design decisions and activities have already taken place, nor is there an account of the reasons why decisions had been made. As a consequence, “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” [28, p. 418]. Furthermore, due to the tentative nature of many design decisions, it may be necessary to return to an earlier stage of the design process for trying other design alternatives. In this case, historical information facilitates the choice of design decisions to undo.

Therefore, information about the history of the design process and the rationale behind decisions is part of the design model in MIKE in addition to a description of the artifact. To that end, DesignKARL introduces an operator which allows to refer to different versions of the model. Furthermore, DesignKARL provides language primitives to express individual design decisions such as, e.g., the choice of a particular data structure. Four basic types of design decisions have been identified. A *realise* design decision captures the refinement of either algorithms (including declarative specifications) or data structures (including KARL language primitives) through more implementation-oriented constructs. For instance, a class and a predicate defining an ordering on the elements of the class can be refined into a sequence of elements of the respective class which captures the ordering directly. The *realise* expression then describes the relationship between the former and the refined representation as an invariant. A second type of design decisions affects the *structure* of some part of the artefact. For instance, an elementary inference action can be turned into a processing module by introducing additional control information because of efficiency considerations. Likewise, a *structure* design decision is also used if the interface of a module is modified. An *introduce* design decision signifies that a new construct is introduced while an *abandon* design decision signifies the inverse, namely, that a construct has been removed from the model.

With these language primitives, it is possible to express which steps are performed during the design process. However, there is no indication yet *why* these steps were carried out. As has been outlined in the previous section, a model construction step is viewed as the gradual decomposition of a requirement until a collection of design decisions can be identified which contribute to meet the top-level requirement. Conversely, the goal decomposition can be interpreted as the justification for the design decisions taken in that particular step. Thus, in order to document the rationale behind design decisions, DesignKARL provides additional language primitives for the description of requirements and their decomposition. In particular, the description of a requirement may refer to a measure which allows to estimate the difference of current and desired status of the requirement.

In addition to the fact that a requirement may be decomposed to a collection of other requirements, various other relationships exist between requirements. As goals may be decomposed in various ways, the designer has to select one of the available alternatives which seems to be most suitable in the given context. The reasons for preferring one alternative over another can be expressed by means of *preferences*. Preferences can only relate nodes in the AND/OR tree which have the same ancestor and which constitute alternative goals. Preferring one alternative over another involves comparing alternatives with respect to criteria which usually are based on the measures associated with the respective requirements. Returning to our previous example, a representation of the collection of schedules as an ordered list would be preferred over a representation as an unordered set since in the former case, the identification of the most recent schedule can be reduced to linear effort (if a new schedule is always put at the beginning of the list) in contrast to an effort of O(n logn) in the second alternative. 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. DesignKARL encompasses additional language primitives to express such *implications* and *exclusions* between subgoals. In the scheduling example, for instance, the reduction of storage needs by recording only the most recent schedule is excluded for reasons of explainability (see also figure 2). Finally, the basic quality of interdependencies between requirements can be described by means of *correlations*. Correlations indicate if actions taken to satisfy one requirement positively or negatively affect the fulfilment of another requirement. For instance, efficiency with respect to processing time and efficiency with respect to storage space are in general inversely *correlated*.

CONCLUSION

An important factor for the improvement of the quality of a software product is the reduction of the overall complexity which the developers have to master. High complexity increases the potential for errors. In MIKE, the attempt to reduce complexity has several facets. First of all, the separation of the development process into different phases reduces complexity since each step can emphasize one particular aspect while neglecting others. In particular, in the design phase in MIKE, attention can be restricted to the question how non-functional requirements determine the actual realisation of the system as the fulfilment of functional requirements has already been ascertained in the knowledge acquisition phase. Yet, life-cycle-based approaches are sometimes criticized because they lack the capability to provide immediate feedback. In order to account for this aspect, MIKE tightly integrates prototyping in its life-cycle. During knowledge acquisition, prototyping has a mostly explorative flavour whereas emphasis changes to experimental prototyping in the design phase. In addition to evaluating design decisions by means of prototyping, evaluation can also be based on quantitative measures for non-functional requirements.

Due to the possibility to impose proper structure on the artefact using DesignKARL's structuring primitives, not only the complexity of the design process, but also the complexity of the design product can be reduced to more manageable pieces.

The transparency of the design process is also improved by the possibility to explicitly describe design decisions and relate them to the requirements that constitute their justification. The model adopted for treating non-functional requirements and describing design rationale is to some extent based on ideas developed in [13, 14] and [28, 29], respectively. Besides making the treatment of non-functional requirements more transparent, the explicit description of design decisions also ensures traceability of functional requirements since design decisions connect parts of the implemented system to those sections of the model of expertise from which they are derived. In addition, the hyper model [4] links these sections in the model of expertise back to the knowledge protocols where the corresponding requirements were expressed informally by the expert. Clearly, the explicit documentation of design process and rationale poses an additional burden on the designer. However, the granularity of the description is in the choice of the designer. Furthermore, this description establishes a basis for providing automated support of the design process itself in a similar way as in, e.g., [26], thus relieving the designer from some low-level tasks. Yet, this aspect has not been addressed in MIKE so far.

Currently, work on a prototypical software environment supporting the design phase in MIKE is under way. Furthermore, our design approach and DesignKARL are used to (manually) design a system for the configuration of elevators.

Acknowledgement

The discussions with Jürgen Angele, Dieter Fensel, and Rudi Studer and their comments on drafts of the paper are gratefully acknowledged.

References

1. J. Angele and D. Fensel: A spiral model for knowledge engineering. Research report 245, Institut für Angewandte Informatik und Formale Beschreibungsverfahren, University of Karlsruhe, 1992.
2. J. Angele, D. Fensel, D. Landes, S. Neubert, and R. Studer: Model-Based and Incremental Knowledge Engineering: The MIKE Approach. In *Knowledge Oriented Software Design*, J. Cuena, ed. IFIP Transactions A-27, Elsevier, Amsterdam, 1993, 139-168.
3. D. Landes and R. Studer: The design process in MIKE. In *Proceedings of the 8th Knowledge Acquisition for Knowledge-Based Systems Workshop KAW'94* (Banff, Canada, January 30 - February 4), 1994.
4. S. Neubert: Model construction in MIKE. In *Knowledge Acquisition for Knowledge-Based Systems,* N. Aussenac et al., eds. Lecture Notes in Artificial Intelligence 723, Springer, Berlin, 1993, 200-219.
5. G. Schreiber, B. Wielinga, and J. Breuker, eds.: *KADS - A Principled Approach to Knowledge-Based Systems Development*. Academic Press, London, 1993.
6. B.J. Wielinga, A.Th. Schreiber, and J.A. Breuker: KADS: A modelling approach to knowledge engineering. In *Knowledge Acquisition 4(1)*, 1992, 5-53.
7. J. Angele, D. Fensel, and R. Studer: The model of expertise in KARL. In *Proceedings of the 2nd World Congress on Expert Systems* (Lisbon/Estoril, Portugal, January 10-14), 1994.
8. D. Fensel: The knowledge acquisition and representation language KARL. Doctoral dissertation, University of Karlsruhe, Germany, 1993.

9. J. Angele: Operationalisierung des Modells der Expertise mit KARL. Doctoral dissertation, University of Karlsruhe, Germany, 1993 (in german).
10. C. Floyd: A systematic look at prototyping. In *Approaches to Prototyping*, R. Budde et al., eds. Springer, Berlin, 1984.
11. B.W. Boehm: A spiral model of software development and enhancement. In *IEEE Computer 21*, 1988, 61-72.
12. D. Landes: Development of knowledge-based systems on the basis of an executable specification. In *Expertensysteme '93*, F. Puppe and A. Günter, eds. Springer, Berlin, 1993, 139-152 (in german). English version available as research report 265, Institut für Angewandte Informatik und Formale Beschreibungsverfahren, University of Karlsruhe, 1993.
13. L. Chung: Representation and utilization of non-functional requirements for information system design. In *Advanced Information Systems Engineering*, R. Andersen, J.A. Bubenko, A. Sølvberg, eds. LNCS 498, Springer, Berlin, 1991, 5-30.
14. J. Mylopoulos, L. Chung, and B. Nixon: Representing and using non-functional requirements: a process-oriented approach. In *IEEE Transactions on Software Engineering 18(6)*, 1992, 483-497.
15. G. Guida and G. Mauri: Evaluating performance and quality of knowledge-based systems: foundation and methodology. In *IEEE Transactions on Knowledge and Data Engineering 5(2)*, 1993, 204-224.
16. D.O. Williams, C. Tomlinson, C.K. Bright, and T. Rajan: The CommonKADS quality viewpoint. Technical report KADSII/T2.2/TR/LR/0040/1.0, Lloyd's Register, London, 1992.
17. G. Schreiber, H. Akkermans, and B. Wielinga: 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), 1990.
18. D. Harel: Dynamic logic. In *Handbook of Philosophical Logic Vol. II*, D. Gabbay and F. Guenthner, eds. Reidel, Dordrecht, 1984, 497-604.
19. J. Reichardt: Preventative software engineering. In *Software Engineering - ESEC'93*, I. Sommerville and M. Paul, eds. Lecture Notes in Computer Science 717, Springer, Berlin, 1993, 251-262.
20. D.N. Card with R.L. Glass: *Measuring Software Design Quality*. Prentice Hall, Englewood Cliffs, 1990.
21. D.L. Parnas: On the criteria to be used in decomposing systems into modules. In *Communications of the ACM 15(12)*, 1972, 1053-1058
22. C. Batini, G. Di Battista, and G. Santucci: Structuring primitives for a dictionary of entity relationship data schemas. In *IEEE Transactions on Software Engineering 19(4)*, 1993, 344-365.
23. P. Jaeschke, A. Oberweis, and W. Stucky: Extending ER model clustering by relationship clustering. In *Proceedings of the 12th International Conference on the Entity-Relationship Approach ERA´93* (Dallas, Texas, December 15-17), 1993.
24. E. Yourdon: *Modern Structured Analysis*. Prentice-Hall, Englewood Cliffs, 1989.
25. G. Schreiber: Operationalizing models of expertise. In [5], 119-149.
26. B.H. Far, T.Takizawa, and Z. Koono: An SDL-based expert system for automatic software design. In *SDL'93: Using Objects*, O. Færgemand and A. Sarma, eds. Elsevier, Amsterdam, 1993, 399-410.
27. D. Landes, D. Fensel, and J. Angele: Formalizing and operationalizing a design task with KARL. In *Formal Specification of Complex Reasoning Systems*, J. Treur and T. Wetter, eds. Ellis Horwood, New York, 1993, 105-141.
28. C. Potts and G. Bruns: Recording the reasons for design decisions. In *Proceedings of the 10th International Conference on Software Engineering* (Singapore, April 11-15), 1988, 418-427.
29. J. Lee: 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), 1991, 114-125.
30. I.D. Baxter: Design maintenance systems. In *Communications of the ACM 35(4)*, 1992, 73-89.