DesignKARL - A Language for 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

Model-based approaches to the development of knowledge-based systems require appropriate description formalisms for representing intermediate results of the development. In the MIKE approach, several such formalisms have been developed spanning the range from the informal specification of requirements to a representation focusing on the realization of the system. In this paper, one of these formalisms, DesignKARL, is presented in some detail. DesignKARL is intended to support the transition from knowledge acquisition to implementation, i.e. the design phase. Besides language primitives for describing the design product, DesignKARL also includes primitives for documenting design decisions and the underlying rationale.

1 Introduction

In recent years, problems in conjunction with developing knowledge-based systems (KBS) by rapid prototyping led to a paradigm shift towards model-based KBS development. A distinctive feature of model-based approaches is an additional description of the problem-solving behaviour at an adequate level which is more abstract than the formalism used for implementing the system [1]. Such an abstract description is missing in rapid prototyping. MIKE [2] is a model-based approach, which, as a lesson from conventional software engineering, also aims at reducing the complexity of the development process by splitting it into different life-cycle phases, each focusing on a particular aspect of the overall development. The results of each phase are described in a distinct model. In particular, the knowledge acquisition phase results in a so-called *model of expertise*. For its description, the formal and operational specification language KARL (cf., e.g., [3], [4]) has been devised in MIKE. In contrast to specifications in conventional software engineering, the model of expertise cannot just state *what* functionality the system must supply, but must also address *how* the required functionality can be exhibited since knowledge about how to solve a problem constitutes an indispensable part of the expertise. However, aspects concerning the *realization* of the functionality on a computer are neglected during knowledge acquisition. These aspects are the concern of the ensuing design phase which results in another model, the *design model* (cf., e.g., [5], [6]).

This paper outlines DesignKARL as the formalism for describing the design model in MIKE. DesignKARL extends KARL by language primitives which particularly aim at facilitating the mapping of an abstract logic-based description of the problem-solving behaviour to an implementation environment. Due to the proximity of the formalisms, a smooth transition from analysis to design is possible in a similar way as it is claimed for object-oriented analysis and design approaches.

Since the model of expertise already covers the functional requirements, the design phase can focus on tackling non-functional requirements, i.e. requirements which influence the shape of the system or the way it exhibits its behaviour rather than describing what the required behaviour actually should be [7]. Similar to conventional software engineering, important issues in this context are the definition of suitable data structures and algorithms and the definition of an appropriate system structure. This paper focuses on the former issues. DesignKARL's structuring primitives are discussed in more detail in [8].

There are indications that, like in conventional software engineering, the design phase is also decisive for the quality of a KBS. In order to reduce the potential for design errors, the design phase should be as transparent as possible. One way to achieve this is thorough documentation of design decisions and their underlying rationale. In MIKE, design decisions are viewed as contributions to the satisfaction of non-functional requirements. Thus, the rationale of a design decision is its relationship to such requirements. Consequently, DesignKARL is not only used to record the shape of the system design (i.e., the design product), but also for describing design decisions and their relationship to affected parts of the design product as well as to the requirements that motivate the decisions.

The following sections presents some of the language primitives of DesignKARL. To that end, we first sketch the ba-

This paper is an extension of an earlier paper which appeared in the *Proceedings of the 6th International Conference on Software and Knowledge Engineering SEKE'94* (Jurmala, Latvia, June 20-23), 1994, 78-85.

sic elements of KARL in section 2 before we outline extensions provided by DesignKARL for use in the design phase in section 3. Section 4 touches on the language primitives for documenting the design process. Section 5 relates our approach to proposals addressing similar aspects and provides an outlook on current and future activities with respect to DesignKARL.

More details about DesignKARL can be found in [9].

2 KARL

The model of expertise in MIKE (and consequently KARL as the formalism for its description) is influenced by central ideas of KADS (cf., e.g., [10]). Specifically, the model of expertise consists of three layers, each of which is concerned with a particular kind of knowledge: domain-specific knowledge at the domain layer, knowledge about inference steps at the inference layer, and knowledge about the control over these inference steps at the task layer.

KARL provides language primitives for specifying the required knowledge at these three layers which resemble the primitives of (extended) entity relationship models, data flow diagrams, and structured program flow diagrams. 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 expressed as *predicates*. Concepts, i.e. collections of entities sharing the same features, are described by *classes* which in turn may be characterized by attributes and which can be arranged in a generalization hierarchy. Any object must be an element of at least one class. An object inherits all attributes defined for the extension of the classes it is associated to. Attributes are either functional or relational. Individual objects and instances of predicates are described by *logical expressions* which are in essence Horn clauses with a few extensions to allow for negation in clause bodies and object-oriented data modelling. The integration of an object-oriented notion into a logic-based framework is accomplished in the same way as in F-logic (cf., e.g., [11]).

The inference layer specifies inference steps by means of *inference actions* and *roles*. Inference actions correspond to processes in a data flow diagram and may be elementary or composed. The behaviour of elementary inference actions is specified declaratively by means of Horn clauses which indicate how output data are computed from input items. Composed inference actions obtain their semantics by decomposition to more basic inference actions and roles. The decomposition of inference actions is accomplished like the decomposition of processes in data flow diagrams. Roles correspond to data stores in data flow diagrams and constitute generic descriptions of inputs and outputs of inference actions. Roles are associated with generic classes and predicates which establish the terminology of the problem-solving method. Three types of roles are distinguished in KARL: *stores*, *views*, and *terminators*. Stores connect inference actions and act as containers for intermediate data. Views and terminators establish upward and, respectively, downward connections to the domain layer by indicating which instances of classes or which tuples of predicates at the domain layer correspond to instances of generic classes or predicates. The connection between items at the domain layer and items at the inference layer is again specified declaratively by means of Horn clauses.

Control over the execution of inference steps is specified declaratively at the task layer using dynamic logic (cf., e.g., [12]). To that end, complex *programs* may be built from primitive programs using the usual control constructs sequence, alternative, and iteration. Primitive programs are either calls of inference actions or assignments to boolean variables. 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. Calls to programs and subtasks at the task layer cause the corresponding inference actions to be executed at the inference layer. More detailed descriptions of KARL may be found in, e.g., [4] and [13].

3 DesignKARL

A model of expertise specified with KARL prescribes what the reasoning component of a KBS must do to solve a given task. Although being executable, this description pays attention to efficiency only to the extent as it determines the problem-solving method, while the issue of an efficient realization of the problem-solving method is part of the design phase. This twofold role of efficiency is discussed in [14] as efficiency at the knowledge level vs. efficiency at the symbol level. As a consequence, functional decomposition (or preliminary design) is already part of the knowledge acquisition phase in MIKE. Therefore, the design phase in MIKE more or less comprises the analogue to detailed design and unit design in conventional approaches, focusing on efficient realization, but also on defining an appropriate system structure in order to facilitate the remaining development activities, supporting maintenance, portability, etc.[1].

Extensions introduced in DesignKARL in addition to KARL primitives address these issues. The use of these additional language primitives (but to some extent also their nature) is constrained by the aim to preserve the structure of the model of expertise as far as appropriate. Structure-preserving design [15] facilitates, e.g., maintenance since portions of design and implementation can be traced back to the model of expertise (or earlier models) more easily. Furthermore, structure-preserving design may support the system´s explanation capabilities since explanations may also resort to more intuitive and less implementation-biased representations.

3.1 Data Types

An object class in KARL characterizes its elements by certain attributes, but abstracts from information which is irrelevant from a conceptual point of view. A key issue in the attempt to improve efficiency is the introduction of data structures which exploit such additional information, e.g. an implicit ordering of elements of a class or of values of a relational attribute, or support data access. Ideally, these data structures should also facilitate the mapping to the implementation environment. To these ends, DesignKARL supplies a collection of data types which currently comprises the types *value*, *class*, *predicate*, and *set* (which are already known in KARL) as well as the types *sequence*, *stack*, *queue*, *n-ary tree*, *hash table*, *index structure*, and *reference*. Instances of these data types can be introduced both at the domain layer and in roles at the inference layer. Each data type is associated with pre-defined operations that may be performed on instances of the respective type for testing, data retrieval, and data storage. For instance, the data type sequence can be manipulated using operations which retrieve the first element of the sequence (*head*), return the sequence without its first element (*tail*), retrieve the data element located at a particular position of the sequence (*[ ]*), insert a data element into a particular position of the sequence (*insert*), append a sequence to another one (*append*), or test if a particular data item appears as an element of a sequence ($\in$, *index*).

The basic idea behind the collection of data types supplied in DesignKARL is to provide means to explicate and exploit inherent ordering relationships in the data (e.g., by means of the data types sequence, stack, queue, or n-ary tree) as well as to be able to efficiently retrieve data from large data sets (e.g., by means of index structures and hash tables). DesignKARL only provides a restricted set of data types since a balance between simplicity and expressiveness has to be met: on the one hand, the data types should be powerful enough to express the designer's intentions, on the other hand, a small collection of data types facilitates the largely automatic generation of appropriate code fragments which implement the respective data types for particular target environments. Data types available in DesignKARL are similar to those in common programming languages like, e.g., C++ or Modula, plus data types, such as hash tables and index structures, which are widely used in database applications. Therefore, the collection of data appears to be sufficient for most applications.

Besides supplying additional data types in DesignKARL, data types which are already known in KARL are extended with additional features. Classes, for instance, can be associated with methods, i.e. user-defined operations. Methods may apply to the class as such (class methods) or to its elements (member methods) and can be specified either by means of logical expressions or by algorithms (see below). Subclasses and instances of a class inherit the methods defined in that class. In DesignKARL, *abstract classes* (cf. e.g. [16]) may be introduced as 'artificial' superclasses in order to exploit method inheritance. Methods are used, e.g., for computing values of derived attributes of an object, but can also be employed in views and terminators for connecting classes at the inference layer with data items at the domain layer. Since a connection between domain and inference layer may also be established via predicates in views and terminators, methods may also be associated to predicates.

References are a means for indirect manipulation of objects or instances of data types. A reference points to the location of the object or instance rather than referring to the object itself. References are convenient if direct manipulation of data items would involve, e.g., 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 data items in the usual way. *Hash tables* and *index structures* are particularly suited for efficient access to large quantities of data (cf., e.g., [17], [18]). The hash function to be used must be defined when declaring a hash table and uses a particular feature of the data items as an argument. *Index structures* exploit a direct association between values of a particular feature of the data items and a location within the instance of the data type used for storing them.

Example 1 Data Types

As an example, consider a simple scheduling task [19] which consists of assigning activities to time slots, taking some restrictions into account, which, e.g., ensure that two particular activities must not be assigned to the same time slot. A collection of associations of activities to time slots constitutes a schedule. A solution to this problem has been specified with KARL which performs three basic steps

1. Although, e.g., the user interface is of paramount importance for the success and acceptance of a KBS, the design model in MIKE currently concentrates on the reasoning mechanisms and the knowledge base since development issues concerning components such as the user interface are already much better understood through work in conventional software engineering. Clearly, due to the interdependence of reasoning component and user interface, the design of the two components should be carried out in parallel. In a later stage of the MIKE project, components such as the user interface will also be covered in the design model.

[20]: first, a new association of an activity to a time slot is proposed and added to the current schedule. In a second step, the resulting schedule is checked against the given restrictions. In case a violation of some restriction is detected, attempts to fix the inconsistency are made in the third step. To that end, a previous association is modified on the basis of the particular inconsistency encountered. After applying a fix, the resulting schedule is checked and, if necessary, fixed again until all restrictions are met. The incremental construction of a schedule according to these three basic steps is repeated until either each activity is associated to a time slot or an inconsistency is detected which cannot be fixed.

In this method for solving scheduling tasks, care must be taken to avoid getting stuck in a cycle by alternatingly applying two fixes, which mutually undo each other's the effects. In order to be able to detect potential cycles, information about previous schedules must be retained, namely in a class *schedule* which is associated to the store *schedules* at the inference layer.

STORE *schedules*
    CLASS *schedule*
        ATTRIBUTES
            associations: SET OF { *association* };
    END;
END;

The most recent schedule can then be identified through a predicate *current_schedule* (*current_state* in [20]):

PREDICATE *current_schedule*
    curr : { *schedule* };
END;

$\forall x_S$ (*current_schedule(curr:* x_S*)*
$\leftarrow x_S \, \varepsilon$ *schedule* $\wedge \neg$ *succ(*x_S*)* ε *schedule*) .

The above expression states that any element of the class *schedule* is considered as current schedule if there is no successor of that schedule in the class *schedule*. Since any element of the class *schedule* may have at most one successor, the current schedule is uniquely characterized by this logical expression. Clearly, this is a compact way of expressing the intended meaning, but the computation of the most recent schedule is fairly expensive as for each schedule all others have to be examined to see if they are the successor of the one in question. The worst case effort[2] of O(n log(n)) may finally be reduced to constant effort if an additional data structure is introduced which explicitly records the linear succession of the schedules in the store *schedules*:

2. The effort of O(n log(n)) in contrast to O(n^2) as one might expect is a consequence of the set-oriented evaluation strategy embodied in the operational semantics of KARL [21].

STORE *schedules*
    CLASS *schedule*
        ATTRIBUTES
            associations: SET OF { *association* };
    END;
    SEQUENCE *schedule_history* OF *schedule*;
END;

Provided that schedules are suitably inserted in the sequence *schedule_history* (cf. Example 2), the current schedule can easily be identified as its first element (in particular, by using the pre-defined operation *head* which returns the first element of a sequence):

$\forall x_S$ (*current_schedule(curr:* x_S*)*
$\leftarrow \boldsymbol{x_S \cong}$ ***head(schedule_history)***) . ♦

3.2 Algorithms

Besides defining appropriate instances of data types, the design phase is concerned with the development of suitable algorithms. While knowledge acquisition focuses on specifying which processing steps have to be carried out in principle to solve the given task, this description may need to be refined and, possibly, restructured in order to be realized efficiently. Thus, an algorithm in DesignKARL need not necessarily be a sophisticated algorithm from a textbook, but may simply result from adding control information (which had previously been neglected since it was not required from a conceptual point of view) to the description of an inference action. The description of algorithms may be carried out similar to control flow specifications in KARL, thus giving them an imperative flavour. An algorithm is built from primitive expressions which either assign a value to a variable or invoke an inference action (or another algorithm) and assign the return values to role variables. In the design phase, variables used in an algorithm may be bound to instances of any of the data types available in DesignKARL. Primitive expressions can be combined using the control constructs sequence, iteration, and alternative.

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

Algorithms may be introduced either in the context of methods associated to object classes at the domain and inference layer or in the bodies of inference actions. Algorithms may be subject to decomposition, i.e. may “call” more elementary algorithms as well as enriched inference actions. Algorithms can be introduced at any level of decomposition, i.e. they may substitute composed inference actions and the corresponding subtasks as well as refine elementary inference actions. Typically, declarative specifications of elementary inference actions are refined to algorithms by adding control information while preserving validity. In this context, the fact that an algorithmic description yields a (not necessarily proper) subset of the output values of the original declarative specification, given the same input data, constitutes a sufficient condition for validity. For validity it is not always required to reproduce the set of output values of the original description exactly. Likewise, the inclusion of results is not mandatory at each level of decomposition of the two representations.

It should be kept in mind that modifications resulting in a different problem-solving method should not be design decisions, but should be carried out in 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 Algorithms

Returning to the scheduling task introduced in Example 1, the incremental computation of a schedule involves the determination of the current schedule and its extension by an extra association of an activity to a time slot. Since these operations are part of the problem-solving method, they are specified as inference actions in KARL. In slight modification to the solution presented in [20], this can be accomplished as follows:

INFERENCE ACTION *select_latest_schedule*
PREMISES *schedules;*
CONCLUSIONS *latest_schedule;*
DEFINITIONS
PREDICATE *current_schedule*
curr : { *schedule* };
END;
RULES ...

$\forall x_S\ (\mathit{current\_schedule}(\mathit{curr}{:}\ x_S)$
$\leftarrow x_S\ \varepsilon\ \mathit{schedule} \wedge \neg\ \mathit{succ}(x_S)\ \varepsilon\ \mathit{schedule}\ )\ .$

$\forall x_S\ \forall x_A\ \forall x_{OA}$
$(\ x_S\ \varepsilon\ \mathit{ltst\_schedule} \leftarrow \mathit{current\_schedule}(\mathit{curr}{:}\ x_S)\ )\ .$

END;

INFERENCE ACTION *create_schedule*
PREMISES *selected_association, schedules, latest_schedule;*
CONCLUSIONS *schedules;*
RULES ...

$\forall x_S\ \forall x_A\ \forall x_{OA}$
$(\ \mathit{succ}(x_S)[\mathit{associations}{::}\ \{x_A, x_{OA}\ \}]\ \varepsilon\ \mathit{schedule}$
$\leftarrow\quad x_S[\mathit{as}{::}\ \{x_{OA}\}]\ \varepsilon\ \mathit{ltst\_schedule} \wedge$
$x_A\ \varepsilon\ \mathit{selected\_association}\ )\ .$

$\forall x_S\ \forall x_A\ \forall x_{OA}$
$(\ \mathit{succ}(x_S)[\mathit{associations}{::}\ \{x_A\ \}]\ \varepsilon\ \mathit{schedule}$
$\leftarrow\quad x_S\ \varepsilon\ \mathit{ltst\_schedule} \wedge \neg\ x_S[\mathit{as}{::}\ \{x_{OA}\}] \wedge$
$x_A\ \varepsilon\ \mathit{selected\_association}\ )\ .$

END;

The premises and conclusions of an inference action refer to the roles which provide input to it (premises) and collect data derived by the inference action (conclusions). The semantics of an inference action is the perfect Herbrand model of the facts and rules in the premise roles and the body of the inference action (cf., e.g., [4]).

The inference action *select_latest_schedule* identifies the current schedule within the class *schedule* (associated with the premise role *schedules*, cf. Example 1) using the predicate *current_schedule* (cf. Example 1) and puts it into the class *ltst_schedule* (associated to the role *latest_schedule*).

The inference action *create_schedule* computes a new schedule. The set of associations of this new schedule comprises all associations of the current schedule (i.e., the one in the class *ltst_schedule)* plus an additional association which is supplied by the store *selected_association*[3].

In the design phase, this KARL specification may now be refined into a DesignKARL description using the data structure *schedule_history* introduced in Example 1. The inference action *create_schedule* is refined into an enriched inference action *extend_schedule* and a (very simple) algorithm *update_history* as shown below:

ENRICHED INFERENCE ACTION *extend_schedule*
PREMISES *selected_association, schedules;*
CONCLUSIONS *schedules;*
RULES ...

$\forall x_S\ \forall x_A\ \forall x_{OA}$
$(\ \mathit{succ}(x_S)[\mathit{associations}{::}\ \{x_A, x_{OA}\ \}]\ \varepsilon\ \mathit{schedules}$
$\leftarrow\quad x_S \cong \boldsymbol{\mathit{head}(\mathit{schedule\_history})} \wedge$
$x_S[\mathit{as}{::}\ \{x_{OA}\}] \wedge$
$x_A\ \varepsilon\ \mathit{selected\_association}\ )\ .$

$\forall x_S\ \forall x_A\ \forall x_{OA}$
$(\ \mathit{succ}(x_S)[\mathit{associations}{::}\ \{x_A\ \}]\ \varepsilon\ \mathit{schedules}$
$\leftarrow\quad x_S \cong \boldsymbol{\mathit{head}(\mathit{schedule\_history})} \wedge$
$\neg\ x_S[\mathit{as}{::}\ \{x_{OA}\}] \wedge$
$x_A\ \varepsilon\ \mathit{selected\_association}\ )\ .$

END;

3. In the clauses, the expressions within square brackets refer to attributes of the respective class and their values. Braces denote the set of values of a relational attribute. The second clause in the inference action *create_schedule* is required for the case that the current schedule has no associations yet.

```
INFERENCE ALGORITHM update_history
    PREMISES schedules;
    CONCLUSIONS schedules;
    RULES ...
        schedule_history :=
            append(succ(head(schedule_history)),
                schedule_history) .
END.
```

Extend_schedule computes a new schedule like *create_schedule* did, but refers to the sequence *schedule_history* instead of the class *ltst_schedule*. The algorithm *update_history* adds the newly computed schedule as new first element to the sequence *schedule_history*. Note that this refinement (the specification of control is omitted here) avoids the computation of the extension of the predicate *current_schedule* and the class *ltst_schedule* altogether, thus yielding a valid, but more efficient realization of the functionality specified in the original KARL description. ♦

3.3 Modules

Experience in conventional software engineering shows the need to reduce the complexity of large software systems by decomposing them into manageable, largely self-contained pieces. To that end, DesignKARL introduces modules as structuring primitives [8]. As the design phase should aim at preserving the structure of the model of expertise, DesignKARL provides two types of modules, namely *domain modules* and *processing modules*, which retain the distinction between domain-specific and domain-independent parts of the required knowledge.

Domain modules collect related domain knowledge in a single place. A module may use knowledge defined elsewhere and, conversely, supply knowledge to other modules. Each module is associated with an interface which restricts access to external knowledge: knowledge defined elsewhere may be used only if it is mentioned in the import interface of the module that intends to use it and if another module makes the knowledge available in its export interface. In the module body, additional classes, predicates, and other data structures may be defined. Their extensions and extensions of classes, predicates, and data structures from the import interface are described by means of logical expressions in the module body. A subset of the knowledge may be exported to other modules. The export of a class, predicate, or data structure implies that also its extension will become known to the importing module. Elements of object classes and tuples of predicates cannot be exported selectively. The rules defining the extension are not accessible, but determine the semantics that must be respected by the importing modules.

Composed inference actions and subtasks in KARL establish a means for functional abstraction at the inference and task layer. A *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).

Processing modules interact with modules at the same level of abstraction by exchanging data via roles, but may also call modules which are part of their decomposition. Like domain modules, processing modules communicate through interfaces which signify, e.g., which roles serve as input or output roles for the inference action described by the processing module as well as which domain modules supply domain knowledge to views and receive knowledge through terminators. Roles which appear in the body of a module, but not in its interface, are not accessible to other processing modules at the same level of decomposition. Roles in the body may, however, be used as input stores or output stores when calling more elementary processing modules. Processing modules may be parameterized with respect to roles and associated classes and predicates and be instantiated as needed at various places.

4 Modelling the Design Process

In addition to a description of the design product, i.e. the artefact that should finally be implemented in a specific manner, a description of the design process, which clarifies which design decisions were taken for which reasons, proves to be valuable for further development and maintenance. During ongoing development, it helps the designer to avoid pursuing unpromising design alternatives repeatedly, but it also facilitates maintenance by improving the understandability of the design product. A description of the design process also supports traceability if it links parts of the design product to the portions of the specification they were derived from and to the requirements that influenced design decisions in that derivation.

In order to achieve these potential benefits, DesignKARL comprises additional language primitives to describe the design process. In principle, the concepts and relationships required to model the design process could be described in a similar way as ordinary domain knowledge. However, since these concepts and relationships have a specific meaning and convey meta information *about* parts of the design product, they are expressed as language primitives in their own right in DesignKARL. Note that only the design product, but not the meta model describing the design process will be implemented in the target environment.

In our model, the design process is viewed as a succession of states of the design product which reflect its evolution from the initial KARL specification into the “final” DesignKARL design description. From a meta level point of view, such a state is a collection of interrelated syntactical objects such as classes, roles, inference actions, modules, etc. In order to be able to address a particular state or its constituents, a so-called *tense-shift operator* @ is introduced [22].

The transition between two adjacent states is effected by activities of the designer, i.e. collections of design decisions. A design decision causes some of the syntactical objects to be substituted by corresponding, but modified syntactical objects in the following state. If the designer takes a design decision, she does so since a particular goal shall be reached, namely a requirement posed towards the design product shall be met. Thus, the justification for a design decision consists in its connection to the (primarily non-functional) requirements which the design decision helps to meet.

In order to be able to formulate such a model of the design process, DesignKARL provides a language primitive to describe *goals*. Goals may express either non-functional requirements to be met or high-level design alternatives. Among other things, goals are characterized by their type such as, e.g., efficiency, a reference to the affected portion of the design product, and a characterization when the goal is considered to be met such as, e.g., required response time or algorithmic complexity. Furthermore, various relationships exist between goals. For instance, "global" requirements may be decomposed to more basic requirements or design alternatives which must be met in order to satisfy the global requirement. This decomposition is continued until the resulting goals can be achieved by elementary design decisions.

Example 3 Goals

The design activities outlined in the previous examples are motivated by the intention to increase the efficiency of the problem-solving process. With DesignKARL, this can be expressed as a goal:

GOAL *SchedEfficiency*
  INSTANCE OF *Efficiency*;
  DECOMPOSED TO *SchedRuntimeEfficiency*,
      SchedStorageEfficiency;
  DECOMPOSITION TYPE OR;
  REFERENCE COMPOSED INFERENCE ACTION
      ProposeAndExchange;
END;

This description expresses that the goal to increase efficiency of the scheduling process primarily affects the inference action *ProposeAndExchange* which constitutes the top level inference action in the description of the design product. Efficiency can be achieved if the runtime efficiency or the efficiency wrt. storage needs can be improved. The desired improvement of the runtime behaviour can in turn be expressed as a goal:

GOAL *SchedRuntimeEfficiency*
  ACHIEVED BY *SeqIntoStSchedules,*
      DAccessToHistoryIAcreate_schedule, ...;
  REFERENCE COMPOSED INFERENCE ACTION
      ProposeAndExchange;
  DESCRIPTION *"Save effort for computing the current schedule by storing the history of intermediate schedules explicitly in an appropriate data structure."*
END;

The goal *SchedRuntimeEfficiency* sketches a potential way to reach the superordinate goals. The goal *SchedRuntimeEfficiency* can be satisfied without further decomposition by means of a collection of elementary design decisions, which, e.g., introduce an additional instance of the data type sequence into the store *schedules* (*SeqIntoStSchedules*) and adapt the affected inference action accordingly by providing direct access to the history of the scheduling process (*DAccessToHistoryIAcreate_schedule*) (see also Example 4). ♦

Goals may affect each other in the sense that steps in order to reach one goal may make it harder to reach a second one (*correlation*). There may also be exclusions, implications, and preferences between requirements. Likewise, several design alternatives may exist at some point of the development, i.e. a requirement may be satisfied by several alternative collections of design decisions. In addition, the choice of a design alternative may exclude (*exclusion*) or, conversely, imply (*implication*) some of the design alternatives for meeting a different requirement. Likewise, some design alternative may be preferred over another one on the basis of other criteria (*preference*). DesignKARL provides language primitives to model these kinds of relationships between goals.

Furthermore, DesignKARL comprises language primitives to describe elementary design decisions. In particular, four types of design decisions are distinguished, namely design decisions which add information concerning the realisation of a particular portion of the design product (*refinement*), which affect a structuring primitive, e.g., by modifying the interface of a module (*structure*), which introduce a constituent which has no counterpart in previous states (*introduction*) such as, e.g., an error handling routine, or which remove constituents of the design product in its new state (*elimination*). Elementary design decisions indicate which two states of the design product they link. Additionally, design decisions of type *refinement* give a detailed account of how the affected constituents of both states relate to each other. This type of description serves the same purpose as, e.g., the retrieve or abstraction functions that are used in the context of refinements in VDM [23].

Example 4 Elementary Design Decisions

The decision to introduce an additional instance of the data type sequence (cf. Example 1) in order to capture some aspects of the history of the scheduling process explicitly is described as a refinement of the store *schedules*:

REFINEMENT *SeqIntoStSchedules*
PRE STORE $schedules@t_n$
POST STORE $schedules@t_{n+1}$
RULES
$\forall x_S$
$((x_S \,\varepsilon\, schedule \wedge \neg\, succ(x_S) \,\varepsilon\, schedule)@t_n$
$\leftrightarrow$
$(x_S \,\varepsilon\, schedule \wedge$
$x_S \cong head(schedule\_history))@t_{n+1})$.

$\forall x_S$
$((x_S \,\varepsilon\, schedule \wedge succ(x_S) \,\varepsilon\, schedule)@t_n$
$\leftrightarrow$
$(x_S \,\varepsilon\, schedule \wedge succ(x_S) \,\varepsilon\, schedule \wedge$
$\exists\, i, j\, (x_S \cong schedule\_history[i] \wedge$
$succ(x_S) \cong schedule\_history[j] \wedge$
$i \cong j + 1)) @t_{n+1})$.
END.

The first rule expressed that fact a schedule without successor will be the first element of the sequence *schedule_history* in the new state t_{n+1} of the design product. The second rule states the fact that in the new state t_{n+1} intermediate schedules are stored in descending order in the sequence *schedule_history*.

Similarly, the adaptation of the inference actions *select_latest_schedule* and *create_schedule* discussed in Example 2 can also be described as a refinement:

REFINEMENT *DAccessToHistoryIAcreate_schedule*
PRE
INFERENCE ACTION $select\_latest\_schedule@t_n$,
INFERENCE ACTION $create\_schedule@t_n$;
POST
ENRICHED INFERENCE
ACTION $extend\_schedule@t_{n+1}$,
INFERENCE ALGORITHM
$update\_history@t_{n+1}$;
RULES
$\forall x_S\, \forall x_A$
$(((x_S \,\varepsilon\, ltst\_schedule \wedge$
$succ(x_S)[associations{::}\, \{x_A\}] \,\varepsilon\, schedule)$
$(\mathbf{select\_latest\_schedule;}$
$\mathbf{create\_schedule}))@t_n$
$\leftrightarrow$
$((succ(x_S) \cong head(schedule\_history) \wedge$
$succ(x_S)[associations{::}\, \{x_A\}] \,\varepsilon\, schedule)$
$(\mathbf{extend\_schedule;}$
$\mathbf{update\_history}))@t_{n+1})$.
END .

This description expresses that, given an equivalent program state in both states t_n and t_{n+1} of the design product, the set of associations of a newly created schedule will be the same after executing the inference actions *select_latest_schedule* and *create_schedule* in the old state t_n on the one hand and after executing *extend_schedule* followed by *update_history* in the new state t_{n+1} on the other hand. Furthermore, the newly created schedule will be at the first position of the sequence *schedule_history* in the new state t_{n+1}. ♦

In a similar way as goals, elementary design decisions may also imply or exclude each other.

Since design activities may be taken in a tentative manner, some of them may need to be revised in a later stage of the design process. Revisions of design activities and of goals can also be modelled with DesignKARL.

5 Conclusion

DesignKARL supports the principled transition from a formal (and executable) specification of the reasoning behaviour of a KBS to its implementation. Approaches to transformational programming (cf. e.g. [24]) pursue a similar goal by providing a largely automatic transformation into the implementation language by provably correct (i.e. semantics-preserving) transformation steps. A less rigorous approach is taken in MIKE as less emphasis is put on the formal correctness of transformation steps (i.e. design decisions and activities) in favour of paying increased attention to non-functional requirements influencing the design process.

It has been claimed as one benefit of transformational programming that the sequence of applied transformations documents the development process. This is similar in spirit to the model adopted for capturing information about the design process in MIKE. The treatment of non-functional requirements is based on ideas developed in the context of information systems design (cf., e.g., [25], [26]). Furthermore, design rationale is described in a similar fashion as in [27] and [28]. The latter proposals promote an issue-based style, which basically consists of setting up a question and providing potential answers. In MIKE, a more result-oriented stance is taken and design decisions are linked to requirements more directly without making an attempt to capture the discourse that leads to the preference of one design alternative over others. MIKE also gives an account of the elementary design decisions which modify the design product. Such a description of elementary design decisions is not normally part of approaches to capture design rationale.

Clearly, the description of the design rationale poses an additional burden on the designer. On the other hand, our approach also establishes the potential for using the description for the provision of intelligent support, i.e. keeping track of requirements to be satisfied, proposing potential targets for design activities, etc. Yet, this area of research has not been addressed in MIKE so far.

Our considerations are less important if a KBS is constructed on the basis of an already available shell since in that case, most design decisions have already 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-purpose shells do not provide a conceptual model to guide knowledge acquisition while role-limiting shells usually lack flexibility if the built-in problem-solving method does not exactly fit the problem at hand.

Current activities focus on the implementation of a support environment, which supports the recording of design decisions and their rationale, and which also allows to execute the developed model as a hybrid prototype, consisting of parts specified with KARL interacting with parts operationalized in the target language C++.

DesignKARL has been used to design a system for the configuration of elevators on the basis of a KARL specification [29].

Acknowledgement

I thank Jürgen Angele, Dieter Fensel, and Rudi Studer for valuable discussions of aspects addressed in this paper.

References

[1] W. Karbach, M. Linster, and A. Voß: Models of problem-solving: one label - one idea? In *Current Trends in Knowledge Acquisition*, B. Wielinga et al., eds. IOS, Amsterdam, 1990, 173-189.

[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. Fensel, J. Angele, and D. Landes: KARL: A Knowledge Acquisition and Representation Language. In *Proceedings of the 11th International Conference on Expert Systems and their Applications* (Avignon, May 27-31), 1991, 513-528.

[4] 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.

[5] 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.

[6] D. Landes: Entwicklung wissensbasierter Systeme auf der Grundlage einer ausführbaren Spezifikation. 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.

[7] D. Landes: Addressing non-functional requirements in the development of knowledge-based systems. In *Proceedings of the 1st International Workshop on Requirements Engineering: Foundation of Software Quality REFSQ'94* (Utrecht, The Netherlands, June 6-7), 1994.

[8] D. Landes and R. Studer: Mechanisms for structuring knowledge-based systems. In *Database and Expert Systems Applications*, D. Karagiannis, ed. LNCS 856, Springer, Berlin, 1994, 488-497.

[9] D. Landes: Die Entwurfsphase in MIKE - Methode und Beschreibungssprache. Dissertation, University of Karlsruhe, forthcoming (in German).

[10] G. Schreiber, B. Wielinga, and J. Breuker, eds.: *KADS - A Principled Approach to Knowledge-Based Systems Development*. Academic Press, London, 1993.

[11] M. Kifer and G. Lausen: F-Logic: A higher-order language for reasoning about objects, inheritance, and scheme. In *Proceedings of the 18th International ACM SIGMOD Conference on Management of Data* (Portland, Oregon, May 31-June 2), 1989, 134-146.

[12] D. Kozen: Logics of programs. In *Handbook of Theoretical Computer Science*, J. v. Leeuwen, ed. Elsevier, Amsterdam, 1990, 789-840.

[13] D. Fensel: The knowledge acquisition and representation language KARL. Dissertation, University of Karlsruhe, 1993.

[14] 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'90* (Banff, Canada, November 4-9), 1990.

[15] G. Schreiber: Operationalizing models of expertise. In [10], 119-149.

[16] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen: *Object-Oriented Modelling and Design*. Prentice Hall, Englewood Cliffs, 1991.

[17] T.H. Cormen, C.E. Leiserson, and R.L. Rivest: *Introduction to Algorithms*. MIT Press, Cambridge, 1990.

[18] R. Elmasri and S.B. Navathe: *Fundamentals of Database Systems*. Benjamin/Cummings, Redwood City, 1989.

[19] I.A. van Langevelde, A.W. Philipsen, and J. Treur: An example reasoning task description. In [30], 7-13.

[20] D. Landes, D. Fensel, and J. Angele: Formalizing and operationalizing a design task with KARL. In [30], 105-141.

[21] J. Angele: Operationalisierung des Modells der Expertise mit KARL. Dissertationen zur Künstlichen Intelligenz 53, infix, St. Augustin, Germany, 1993 (in German).

[22] R. Rönnquist: Theory and practice of tense-bound object references. Dissertation, University of Linköping, 1992.

[23] D. Bjørner: Specification and transformation. Methodology aspects of the Vienna Development Method - Towards a

meaning of 'M' in VDM. In *Formal Description of Programming Concepts*, E.J. Neuhold and M. Paul, eds. Springer, Berlin, 1991, 137-258.

[24] H.A. Partsch: *Specification and Transformation of Programs*. Springer, Berlin, 1990.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

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

[30] J. Treur and T. Wetter, eds.: *Formal Specification of Complex Reasoning Systems*. Ellis Horwood, New York, 1993.