Mechanisms for Structuring Knowledge-Based Systems

Dieter Landes and Rudi Studer
Institut für Angewandte Informatik und Formale Beschreibungsverfahren
Universität Karlsruhe, D-76128 Karlsruhe, Germany
e-mail: { landes | studer }@aifb.uni-karlsruhe.de

In order to reduce the complexity of large knowledge-based systems and promote reusability, means for decomposing them to smaller chunks are required. MIKE, our knowledge engineering framework, provides three basic means for structuring which are described in this paper: different kinds of knowledge are separated at different knowledge layers, knowledge layers can be structured by modules, and knowledge within modules is expressed in terms of an object-centred data model. In addition, ideas from entity relationship model clustering are adapted and extended to facilitate the understandability of domain knowledge and support the formation of modules.

1 Introduction

Two main principles for structuring programs evolved in computer science: modularization and object-orientation. While these mechanisms are commonly employed in "conventional" programs in order to exploit their well-known benefits, the situation is slightly different for knowledge-based systems (KBS). Common knowledge representation formalisms for KBS such as production rules, frames, or semantical networks impose structure on the knowledge they embody by expressing it in terms of concepts or rules, which are comparable to objects (and expressions involving such objects) in object-oriented approaches with respect to granularity. In these representation formalisms, however, usually no additional, more coarse-grained structuring primitives are available which might constitute the counterpart to modules in traditional computer science. Conversely, several proposals have been made for modularizing logic programs or logical theories (cf., e.g., [9], [10], [11]). These proposals usually use modules or contexts as the only structuring primitive, i.e. the contents of such structures are basically unstructured collections of first order logic sentences.

Due to the complexity of KBS in realistic settings, appropriate structuring mechanisms at different levels of granularity are required. For that reason, three types of structuring mechanisms are available in MIKE (Model-based Incremental Knowledge Engineering), our framework for developing KBS [1]. MIKE provides three distinct layers to separate domain knowledge, knowledge on inference steps, and knowledge on the control over these inference steps. This distinction is inspired by an analogous separation in the KADS framework (cf., e.g., [12]) for knowledge engineering. The layer containing domain-specific knowledge (i.e. the domain layer) can be structured further by means of domain modules, while portions of the two other layers (i.e. inference and task layer) can be encapsulated in so-called processing modules. Domain and inference knowledge in modules is expressed primarily in terms of object classes and objects, thus introducing an object-centred notion.

Structuring systems using these three basic mechanisms facilitates understandability since it allows to obtain global overviews as well as to go into the details of relevant parts. Such an organization also supports maintenance since parts of the system that must be adapted due to a changed context are easier to determine. Furthermore, reuse is supported since constituents of the system interact in a controlled way, e.g., via interfaces in the case of modules.

In section 2 of this paper, we will shortly address the three knowledge layers and outline the primitives of KARL ([2], [5], [3]), the formal and executable specification language used for expressing knowledge at these layers in MIKE. The main parts of the paper deal with the two types of modules (section 3) and with so-called clusters (section 4) as additional structuring mechanism which is inspired by work on clustering entity relationship models. Clusters are intended to promote understandability and facilitate the formation of domain modules. Section 5 puts the structuring mechanisms of MIKE into perspective of related work and section 6 concludes the paper. Object classes and objects are not particularly addressed here since they are discussed at length in the context of KARL. Language primitives for describing modules are part of DesignKARL, the design language of MIKE [7]. It should be noted that the structuring primitives of MIKE are not part of a particular implementation environment, but rather indicate how an appropriately structured KBS might be realized within a suitable implementation environment.

2 Knowledge Layers

Like KADS (cf., e.g., [12]), MIKE distinguishes three types of knowledge each of which is kept at a distinct knowledge layer. The domain layer contains domain-specific knowledge while the inference layer comprises knowledge about inference steps that might potentially be performed for solving a problem. Control over these inference steps is not specified at the inference layer, but at the task layer. This separation of different knowledge types allows to describe problem-solving methods (i.e. inference plus task layer) in a generic and domain-independent way, thus making it possible to reuse problem-solving methods in a different domain or, conversely, reuse (parts of) a domain model for a different application and problem-solving method. The description of the problem-solving method is accomplished at two different layers as experience showed that maintenance may become very complicated if control knowledge is intertwined with the knowledge what the inference steps actually are.

Knowledge is described in MIKE using the declarative specification language KARL [2] which is intended to be used during knowledge acquisition, i.e. focuses on conceptual issues. Therefore, KARL provides epistemological primitives which allow to model the knowledge of an expert precisely without immediately casting it into the peculiarities of a particular representation formalism. To that end, KARL uses primitives which resemble the primitives of (extended) entity relationship models, data flow diagrams, and structured program flow diagrams. Domain knowledge is expressed basically in terms of *objects* (denoting individual entities in the domain of discourse together with their properties), *classes* (denoting concepts), and *predicates* (describing relationships between entities). Additionally, sufficient conditions for properties of objects or tuples of predicates can be expressed as Horn clause expressions. Knowledge about inference steps is expressed mainly by means of (elementary or composed) *inference actions* and *roles*. Inference actions correspond to processes in data flow diagrams. Like hierarchical data flow diagrams, composed inference actions can be decomposed, resulting in a collection of roles and more elementary inference actions. The behaviour of elementary inference actions is specified declaratively with Horn clauses. Roles correspond to data stores in data flow diagrams, i.e. provide input to or collect output of inference actions. Roles are associated with class and predicate definitions establishing the terminology of the problem-solving method. Some roles (namely, views and terminators) are connected to the domain layer, thus making domain knowledge accessible to inference actions. Finally, control knowledge is expressed by means of *programs* which are calls to inference actions, assignments to boolean variables, or more complex constructs built by sequence, alternative, and iteration. Programs may be combined to *subtasks*, which constitute the analogue to composed inference actions at the inference layer and define their internal control flow. The connection between task and inference layer is established by the fact that inference actions can be called at the task layer.

3 Modules

The distinction of knowledge layers in MIKE is one step towards decomposing large software systems into manageable pieces. Additionally, MIKE collects data objects sharing the same properties in classes and encapsulates descriptions of properties (i.e. attributes) with the objects they apply to. Still, additional means for imposing structure on a KBS are required between these opposite ends of the spectrum of granularity. Modules are an appropriate additional structuring primitive since they further reduce the overall complexity by splitting a software system into meaningful and manageable pieces while bringing about the advantages of information hiding.

The separation of knowledge layers and the notion of concepts and individual entities is part of the conceptual model underlying MIKE. Therefore, these structuring mechanisms are already present in KARL (and even in a preceding semiformal representation). The definition of modules, however, is a matter of realization, to be addressed during system design, rather than a conceptual issue. Consequently, modules are introduced as a language primitive of DesignKARL [7], which extends KARL by the ability to express realization-oriented aspects. In order to facilitate maintenance, the design phase in MIKE aims at preserving the structure of the model developed in the analysis phase, in particular the distinction between domain knowledge and domain-independent knowledge about the problem-solving method. Therefore, DesignKARL provides two types of modules, namely *domain modules* and *processing modules*.

3.1 Domain Modules

Domain modules collect related domain knowledge in a single place. A module may use knowledge defined elsewhere as well as supply knowledge to other modules. Access to external knowledge, i.e. classes and predicates, is restricted by module interfaces: knowledge defined elsewhere may be used only if it is mentioned in the import interface of the module intending to use it and if another module makes the knowledge available in its export interface. Imported knowledge may be renamed. Additional classes and predicates may be defined in the declaration part. In the body, extensions of classes and predicates from the interface and declaration parts are described by means of Horn clauses or simple facts. The body comprises an additional subsection for facts that are subject to change when the system is solving another case, thus distinguishing case data from knowledge which is constant across different cases. A subset of imported and locally defined classes and predicates may be exported to other modules. Exporting classes or predicates implies that their extension will also 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 (cf., e.g., [10]).

Example 1: [8] report on a KARL specification of a solution to a configuration task, namely configuring elevator systems. A valid configuration of an elevator system consists of a collection of components such that none of the constraints on their compatibility is violated. Knowledge concerning such components might then be collected in domain modules. Knowledge concerning, e.g., the car might be found in the module *car-data* (cf. Fig. 1.) which exports some knowledge to the outside without using knowledge described elsewhere (i.e. the import interface is empty). ♦

```
DMODULE car-data
  INTERFACE
    EXPORT          // Class and predicate definitions to be used elsewhere ...
      CLASS car                               CLASS base
          car_platform:  { platform };            bs_model:  { STRING };
          car_sling:     { sling };               bs_height: { INTEGER };
          car_door:      { door };            END;
          ...                                 ...
      END;                                    PREDICATE base_order
      CLASS platform                              act:     { base };
          pl_base:   { base };                    next:    { base };
          pl_width:  { INTEGER };             END;
          ...                                 ...
  BODY
    DEFINITIONS     // Class and predicate definitions to be used only locally ...
      PREDICATE intermediate_base
          lb:      { base };
          ub:      { base };
      END;
    RULES           // Intensional descriptions of classes and predicates ...
      ∀x_B ∀y_B ∀z_B ∀x_M ∀y_M ∀z_M
        ( intermediate_base(lb: x_B, ub: y_B)
            ←  x_B[base_model: x_M] ε base ∧ y_B[base_model: y_M] ε base ∧
               z_B[base_model: z_M] ε base ∧ x_M < z_M ∧ z_M < y_M ) .
      ∀x_B ∀y_B ∀x_M ∀y_M
        ( base_order(act: x_B, next: y_B)
            ←  x_B[base_model: x_M] ε base ∧ y_B[base_model: y_M] ε base ∧
               x_M < y_M ∧ ¬ intermediate_base(lb: x_B, ub: y_B) ) .
    FACTS           // Extensional descriptions of classes and predicates ...
      base25B[bs_model: "2.5B", bs_height: 6.625] ε base .
      base6B[bs_model: "6B", bs_height: 6.6875] ε base .
    INPUTDATA       // Case-specific data ...
      pltf[pl_width: 70] ε platform .
END;
```

Fig. 1. A domain module

3.2 Processing Modules

The notion of composed inference actions and subtasks in KARL already constitutes a means of abstraction at the inference and task layer. *Processing modules* are based on this notion since each of them describes a composed inference action and its associated subtask, thus resembling procedures in common programming languages. Due to the close relationship of task and inference layer, processing modules collect knowledge of both layers. The body of a module details the decomposition of an inference action into more basic inference steps and roles in the interface part, while the control flow among these inference steps is described in the control part. Thus, the distinction between the two different types of knowledge is still largely retained.

Like domain modules, processing modules communicate through interfaces. 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. 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 problem-solving method and between problem-solving method and domain knowledge base. That is, the interface lists input and output roles of the inference action described by the processing module as well as domain modules which supply domain knowledge to views and receive knowledge through terminators. Roles appearing only in the body of a module are not accessible to other processing modules at the same level of decomposition, may, however, be used as input stores or output stores by more elementary processing modules. Processing modules may be parameterized with respect to roles and associated classes and predicates and can be instantiated as needed at various places.

The functional decomposition carried out during knowledge acquisition implies an initial modularization, yet may not be the best decomposition from a realization point of view. Therefore, this initial decomposition may be subject to modifications during the design phase.

Example 2: A major step in solving the elevator configuration task of Example 1 consists in proposing a yet unknown value of a parameter of the elevator (e.g., the model of the platform base) which is computed on the basis of previously determined parameter values, thus extending the set of known parameter values in the store *KnownParameters*. Knowledge about how to actually compute a parameter is available in domain modules which are accessible through the view *Parameters*. This can be summarized in a processing module as shown in Fig. 2. ♦

4 Clusters

An initial partitioning of task and inference layer into processing modules is based on the functional abstraction brought about by composed inference actions and subtask. For the formation of domain modules, however, related portions of domain knowledge must be identified, e.g. by abstracting from unnecessary detail and thus improving understandability of large models. A similar problem arises when conceptual models in database applications grow so large that the entity relationship (ER) models describing them become unreadable. Several proposals (cf., e.g., [13], [4], [6]) tackle this problem by constructing hierarchies of ER models by means of clustering. The basic idea is to abstract from the internal structure of a portion of an ER model in a new model where

```
PMODULE propose
  INTERFACE
    PREMISES                                              CONCLUSIONS
      STORE KnownParameters                                 STORE KnownParameters
        CLASS KnownParam                                        ...
            value:          { };                            END;
            depends:        SET OF { Param };
        END;
      END;
      VIEW Parameters
        CLASS Param                                       DMODULES
            value:        { };                              elevator-data, car-data, ...
            depends:    SET OF { Param };
          ...
      END;
  BODY
      CONTROL
        (STORES PossibleParameters) :=
          SelectPossibleParameters(STORES KnownParameters, VIEWS Parameters);
                // Determine set of parameters that can currently be computed
        IF  ( ¬ ∅(PossibleParameters, PossibleParam) ) THEN
                // If there is a parameter that can currently be computed ...
            ...
        ELSE end := TRUE;       // Otherwise we are done
        ENDIF;
      INFERENCE
        STORE PossibleParameters
          CLASS PossibleParam ISA Param;
          END;
        END;
        ...
        INFERENCE ACTION SelectPossibleParameter
          PREMISES KnownParameters, Parameters
          CONCLUSIONS PossibleParameters
          ...
          RULES
            ∀x_p ( x_p ε PossibleParam
                ←  x_p ε Param ∧ ¬ x_p ε KnownParam ∧ ¬ unknown_deps(p: x_p) ).
        END;
        ...
  END;
```

Fig. 2. A processing module

the respective sub-model is substituted by a new artificial object, a so-called *cluster*. Since the language primitives that KARL supplies for describing the ontology of an application correspond to those of an extended ER model, ideas from ER model clustering also apply to KARL descriptions of domain layers. Three basic clustering mechanisms are distinguished: *concept clustering*, *property clustering*, and *complex clustering* which largely correspond to entity clustering, simple relationship clustering, and complex relationship clustering in [6]. Notice that clustering does not pay attention to the extension of classes and predicates.

Concept clustering maps related object classes into a higher-order cluster, e.g., by clustering a “dominating” concept with its “dominated” concepts. Concepts forming the range of an attribute may be clustered with the concept forming its domain. In this case, the attribute (which is viewed as a special kind of predicate in this context) is included in the cluster. Likewise, relationships (i.e., predicates) are usually included in such a cluster if they only involve the concept forming the range of the attribute in question. Concept clusters may also be formed by abstracting several semantically similar concepts into a cluster (cf. abstraction grouping in [6]). For instance, subconcepts may be clustered with their common superconcept. Clusters that are formed using concept clustering have the character of concepts and may be used like elementary classes in predicates or when forming even more abstract clusters.

Property clustering abstracts semantically similar relationships into a cluster. Relationships in this context comprise predicates as well as attributes with non-elementary range. Attributes with elementary ranges such as, e.g., strings or integer numbers are not considered in our clustering scheme since they are assumed to be an integral part of the concept constituting their domain. Thus, such attributes may simply be neglected as a first step of abstraction. Property clusters may either collapse only predicates or only attributes into a cluster which then behaves as an abstract predicate or abstract attribute. Before applying property clustering the classes or clusters involved in the relationships or constituting the domains and ranges of the affected attributes must be clustered by concept clustering. This treatment of attributes extends the proposals in [13], [4], or [6].

Finally, complex clustering allows to collapse parts of domain ontology into a cluster which cannot be clustered according to the concept or property clustering schemes, but still bear sufficient similarities. Depending on its contents, the resulting cluster may have the character of either a class or a predicate.

Example 3: Returning to the elevator configuration problem of the previous examples, a small part of the domain ontology (which comprises a total of some 50 classes and 40 predicates) is shown in the graphical notation of KARL (which is quite similar to an ER diagram) in Fig. 3. Due to the complexity of the complete model, clustering is useful in order to abstract from some of the details.

Concept clustering can be applied to cluster the platform base with the platform of the

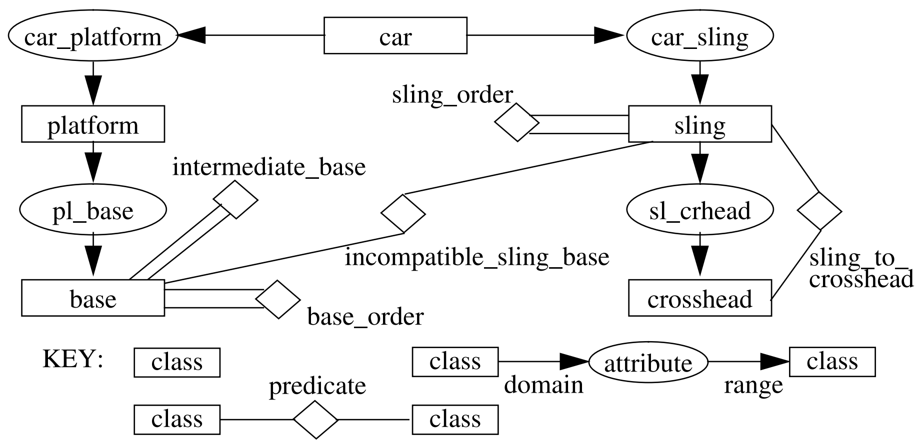

Fig. 3. Part of the domain ontology for an elevator configuration task

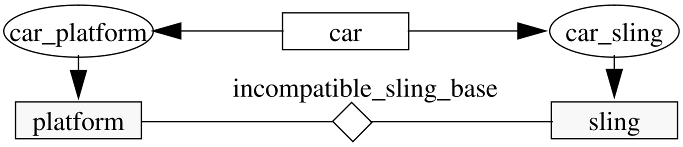

Fig. 4. Part of the domain ontology after introducing two clusters *platform* and *sling*

elevator. The predicates *base_order* and *intermediate_base* and the attribute *pl_base* are also included in the newly formed cluster. Since *platform* is the dominating class, the cluster will be called *platform* as well. Since the predicate *incompatible_sling_base* refers to the platform base as one of its arguments, this reference has to be replaced by a reference to the cluster *platform* since the class *base* is encapsulated in the respective cluster and no longer visible in the abstract representation (cf. Fig. 4.). Complex clustering is applied to form a second cluster, *sling*, comprising the classes *sling* and *crosshead*, the attribute *sl_crhead*, and the predicates *sling_order* and *sling_to_crosshead*.

Clustering might be carried even further by collapsing the clusters *platform* and *sling* (and other constituents of an elevator's car which are not shown in Fig. 4.) into a new cluster *car_components* (abstraction grouping) and by forming an attribute cluster *components* from the attributes *car_platform*, *car_sling* etc. (property clustering). ♦

Especially when forming clusters using complex clustering, links between a cluster and its environment, i.e. all classes, attributes, or predicates to which the cluster is connected, must be consistent with the links in more detailed representations. For instance, it is illegal to form a predicate-like cluster in such a way that it is linked to another predicate. Furthermore, arguments of predicates as well as domains and ranges of attributes must be adapted when forming clusters. In the previous example, the link of the predicate *incompatible_sling_base* to the class *base* must be changed into a connection to the cluster *platform*. The transformation of a cluster into its refinement and vice versa is expressed quite similar as in [4] by indicating which items in the abstract representation are replaced by which items in the refinement and vice versa.

Example 4: Taking the cluster *platform* from the previous example, the relationship between the cluster and its refined representation is specified by the transformation below:

```
CLUSTER platform
    ABS    platform, PRED(incompatible_sling_base, platform, sling);
    SPEC   platform, base,
           PRED(incompatible_sling_base, base, sling), PRED(base_order, base, base),
           PRED(intermediate_base, base, base), ATT(pl_base, platform, base);
END;
```

The classes *platform* and *base*, the predicates *base_order*, *intermediate_base*, and *incompatible_sling_base*, and the attribute *pl_base* are removed from the model while the class (cluster) *platform* and the predicate *incompatible_sling_base* (with modified first argument) are added in the abstract view. If the internals of the cluster are to be inspected the transformation is applied in the reverse direction, i.e. the items in the *SPEC* section are replaced for the constituents in the *ABS* section. ♦

In contrast to modules, which prescribe a partitioning of the implemented system (provided the implementation environment includes a possibility to define modules), clustering is a means to improve understandability during development, but has no direct counterpart in the implementation. Clustering is useful early in the design process since clusters may indicate which parts of knowledge might be candidates for encapsulation in the same module. The clusters defined in Example 3, e.g., indicate that their contents should be collected in one module (cf. Example 1).

5 Related Work

[14] present an approach to structuring knowledge bases into three knowledge layers. Layer H1 contains factual data on the basis of which layer H2 draws inferences. Pragmatic guidance of when to draw which inferences is provided by layer H3 which embodies heuristics which hold in particular applications. Layer H1 partly corresponds to the domain layer in MIKE, layer H2 is the counterpart of the inference layer, and layer H3 can be viewed as an analogue to the task layer with an additional strategic component. The layers H1 and H2 do not coincide completely with the domain and inference layers of MIKE since H1 only contains factual domain data and H2 is not generic in the sense of the MIKE inference layer, but also contains domain specific knowledge in addition to particular inferential capabilities. Therefore, layer H2 roughly corresponds to the MIKE inference layer plus the part of the domain layer comprising case-independent knowledge.

Each layer is composed of sub-structures. H1 usually consists of several autonomous data bases, whereas layer H2 contains so-called SoDs. Layer H3 may consists of several applications. Basically, an application consists of an appropriate collection of SoDs which in turn establish suitable views on the data at the bottom layer. [14] focus on SoDs which, in principle, correspond to processing modules. In contrast to processing modules which describe a complex inference step and its internal control flow and which define a method-specific ontology, SoDs are larger granules. The main problems of this decomposition scheme are the missing separation of different types of knowledge, thus making reuse of parts of the system more difficult, and the fairly large granularity of SoDs, which still leaves the need for smaller constituents which are more easily understood.

In [11], a (logic) knowledge base is viewed as a theory which is developed from a generic kernel by applying construction operators, i.e. specific theory morphisms. These theory morphisms correspond to the semantic primitives of the chosen knowledge representation approach. Thus, each application of such a theory morphism introduces an instance of the corresponding representation primitive into the knowledge base. Intermediate theories may serve as the building blocks that can be reused across applications. However, these reusable chunks are not modules in the usual sense since they do not have an explicit interface that protects some of the chunk's contents from unwanted access. Furthermore, it is not clear if this type of organization of the knowledge base can actually be exploited efficiently since the computation of the construction steps which is required for arriving at a particular stage in the development of the global development (i.e. a particular intermediate theory or "module") can be quite costly.

6 Discussion

Three different kinds of structuring mechanisms are used in MIKE already during KBS development. First, knowledge is represented at three different knowledge layers, thus separating domain knowledge from the generic problem-solving method. Second, knowledge layers can be structured further by means of domain modules and processing modules. Third, the contents of modules are expressed in terms of an object-centred data model. Through this organization, reuse of knowledge is supported at various levels of granularity. Furthermore, understandability of the overall system is improved since attention can be focused on particular, largely self-contained parts of the system. The understandability of domain knowledge is facilitated further by the possibility to abstract from details by means of clustering.

Further work is required to develop stronger guidelines which clusters should be formed and what their contexts should be, in particular when complex clustering has to be employed. Currently, a tool supporting the formation and manipulation of clusters is under development. In addition, the semantics of KARL ([5], [3]) is extended to cover the module concept outlined informally in this paper. This might also improve the efficiency of the KARL interpreter since the perfect Herbrand models constituting the semantics of inference action only need to include the relevant domain modules instead of the complete domain knowledge. The framework presented in this paper is applied quite successfully to the elevator configuration problem [8] mentioned in the examples.

References

[1] 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.

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

[3] J. Angele: Operationalisierung des Modells der Expertise mit KARL (Operationalization of the model of expertise with KARL). infix Verlag, St. Augustin, Germany, 1993 (in german).

[4] C. Batini, G. Di Battista, and G. Santucci: Structuring primitives for a dictionary of entity relationship data schemas. In *IEEE Trans. on Software Engineering 19(4)*, 1993, 344-365.

[5] D. Fensel: The knowledge acquisition and representation language KARL. Doctoral dissertation, University of Karlsruhe, Germany, 1993.

[6] P. Jaeschke, A. Oberweis, and W. Stucky: Extending ER model clustering by relationship clustering. In *Proc. 12th Int. Conf. on the Entity-Relationship Approach ERA´93* (Arlington, Texas, Dec. 15-17), 1993, 447-459.

[7] D. Landes and R. Studer: The design process in MIKE. In *Proc. 8th Knowledge Acquisition for Knowledge-Based Systems Workshop KAW'94* (Banff, Canada, Jan. 30 - Feb. 4), 1994.

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

[9] U. Pletat: The knowledge representation language L_{LILOG}. In *Text Understanding in LILOG*, O. Herzog and C.-R. Rollinger, eds. LNAI 546, Springer, Berlin, 1991, 357-379.

[10] U. Pletat: Modularizing knowledge in L_{LILOG}. IWBS Report 173, IBM Germany, Stuttgart, 1991.

[11] C. Sernadas, J. Fiadeiro, and A. Sernadas: Modular construction of logic knowledge bases: an algebraic approach. In *Information Systems 15(1)*, 1990, 37-59.

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

[13] T.J. Teorey, G. Wei, D.L. Bolton, and J.A. Koenig: ER model clustering as an aid for user communication and documentation in database design. In *CACM 32(8)*, 1989, 975-987.

[14] G. Wiederhold, P. Rathmann, T. Barsalou, B.S. Lee, and D. Quass: Partitioning and composing knowledge. In *Information Systems 15(1)*, 1990, 61-72.