MODEL-K for prototyping and strategic reasoning at the knowledge level

Werner Karbach and Angi Voß

German National Research Institute for Computer-Science (GMD)
AI Research Division
Postfach 1316
D-5205 Sankt Augustin, FRG
e-mail: karbach@gmdzi.gmd.de; avoss@gmdzi.gmd.de

Abstract. To close the gap between knowledge level and symbol level, the MODEL-K language allows to specify KADS conceptual models and to refine them to operational systems. Since both activities may be arbitrarily interleaved, early prototyping is supported at the highest level. Systems written in MODEL-K contain their conceptual model, making them more transparent, easier to communicate to the expert, to explain to the user, and to maintain by the knowledge engineer.
The strategy layer of KADS is supposed to control and possibly repair the activities being modeled by the lower layers. MODEL-K views this kind of strategic reasoning as a meta-activity. In the REFLECT project, we came to view meta-activities like resource-management or competence assessment as ordinary problem solving methods, that in turn can be described using KADS. Correspondingly, we extended MODEL-K to model and operationalize such meta-activities. In particular, the lower three layers and the system they model are automatically kept consistent due to the construction of MODEL-K[1].

1 Motivation

Combining modeling and rapid prototyping In the development of knowledge-based systems there is a recognizable shift from the traditional rapid prototyping approach to model-based approaches. The need for explicit and higher-level descriptions of problem solving methods arose in knowledge acquisition [40] [20], in attempts to reuse knowledge bases [5] and in research on explanations [22] and tutoring [9]. Newell [23] argued that the "knowledge level" is the right level for these purposes.

All model-based approaches to problem solving fall into one of two categories: Either they provide a universal framework to specify arbitrary problem solving

[1] This work is part of a research project partially funded by the Esprit Basic Research Programme of the Commission of the European Communities as project number 3178. The partners in this project are the University of Amsterdam (NL), the German National Research Institute for Computer-Science GMD (D), the Netherlands Energy Research Foundation ECN (NL), and BSR-Consulting(D).

methods which are not operational, or they provide a specialized but operational framework (for a detailed comparison see [17]). As a representative of the first category, KADS [40] [4] proposed a strict separation between knowledge analysis and implementation. The output of the first phase is a (paper-based) 4-layered conceptual model describing the expertise. To implement such a model, a completely separate design model is to be developed. As a consequence, the model of the analysis phase and the implementation are completely disconnected. Therefore it may be difficult to recognize the model in the system, to transfer modifications from the model to the system or vice versa, and to explain the behaviour of the system in terms of the model.

Representatives of the second category are operational systems having an underlying model of the special task they perform. Examples are SALT, MORE and others described in [20], the generic task languages CSRL for diagnosis [6] and DSPL for design [7], shells like MED2/D3 [25] and the knowledge acquisition tool generator of [21]. As their models are mostly described verbally, it is difficult to compare the suitability of these systems for the problem at hand. Moreover, the systems can hardly be adapted or combined, since the paradigms are different and the knowledge level model is not exactly reflected.

What we need is both, a framework which allows us to conceptualize our very first ideas on how to approach an arbitrary problem, and to refine these ideas stepwise to the extent as our pervasion of the problems grows until all operational details are filled in. Thus an operational model is obtained by successive refinements of the conceptual description. This allows prototyping cycles as early as any partial model is executable.

The idea of operational descriptions is not completely new. In conventional software engineering, the executable specifications serve a similar purpose: a very abstract, formal specification is stepwise refined until an executable algorithmic specification or rewrite system is obtained [3]. The advantages of prototyping in software development are described in [13].

The MODEL-K language to be presented in this article allows to model and operationalize KADS' models of expertise. Several such languages have been put forward by now. ML^2 [33] and FORKADS [38] are logic-based, KARL [12] maps into entity relationship descriptions, logic, and an ordinary control language. OMOS [18], like MODEL-K, translates into BABYLON [8]. Not for KADS, but for the components of expertise approach, [34] developed an architecture on top of KRS that establishes a close correspondence between a knowledge level model and its operationalization. In fact, this approach inspired us to more clearly distinguish between the conceptual model and its operational refinement.

Strategic and meta-reasoning The need for meta-reasoning in knowledge based systems was recognized very early in the AI community [31] [10], [11]. Strategic control, assessing and improving one's own competence, detecting deadends in problem solving, tutoring about, or explaining a knowledge based system are typical meta-activities. Although specific systems like HACKER [31], TERESIAS [11], REASON [28], MOLGEN [30] and PDP [16] were built, a ge-

neral framework for incorporating meta-reasoning into knowledge-based systems is missing.

KADS introduced the strategy layer in its conceptual models in order to cope with this kind of knowledge. However, the notions at this level were so vague that it was hardly ever used and was often mixed up with the task layer. Since the strategy layer dynamically reasons *about*, controls and possibly repairs the lower layers, these layers must be causally connected to their implementation, the so-called object system. A causal connection is a mechanism guaranteeing consistency between the object system and its model. In particular, any progress in the object system must be reflected upwards, and any modifications of the model by the strategic layer must be reflected downwards [19].

An object system built in MODEL-K ideally suits these purposes, since the running system incorporates its own model. Thus consistency between object model and object system is automatically guaranteed by the MODEL-K interpreter.

Other than in computational reflection, the object model consisting of the three lower KADS layers hides any implementation details. They are irrelevant since the kind of meta-reasoning going on at the strategic layer is at the knowledge level [2]. In fact, our work in the REFLECT project [37] [27] revealed that meta-activities like competence assessment and improvement or resource management, although having another computational system as their domain, can be interpreted as generic problem solving methods and can be modeled like any ordinary object level methods. For example, competence assessment can basically be regarded as a diagnosis task.

In the REFLECT project we elaborated these ideas into a framework for "knowledge level reflection". Not only did we extend KADS models of expertise to include a meta-model at the strategy layer, but we also extended MODEL-K to interwovenly specify and implement such meta-models. In fact, the necessary additions were almost straightforward and involved only a few important design decisions.

In the following sections, we will first describe "ordinary MODEL-K" using an office room allocation problem, and later discuss the strategy layer and the meta-level extensions.

2 MODEL-K for object systems

Figure 1 sketches the MODEL-K framework. We distinguish the conceptual model and its operational refinement. Both are described on three layers: domain, inference and control layers. In the conceptual model, the domain layer describes the domain structure in terms of concepts and relations, the inference layer determines the inference structure in terms of knowledge sources, i.e. basic operators and metaclasses, and the control layer specifies the control structure in terms of tasks. In the operational refinement, concepts are supplemented by their instances, relations by their tuples, knowledge sources by their bodies, i.e. executable pieces of code, and tasks by their control statements. For the

connection between inference and domain layer, the same distinction is made with built-in default operational refinements. Although syntactically separated, conceptual modelling and operational refinement can be arbitrarily interwoven during knowledge engineering.

Beside the static description of the problem-solver, MODEL-K provides an explicit representation of the dynamic status of system while problem solving: an agenda contains the pending tasks, and the metaclasses contain the variable data that are manipulated by the knowledge sources

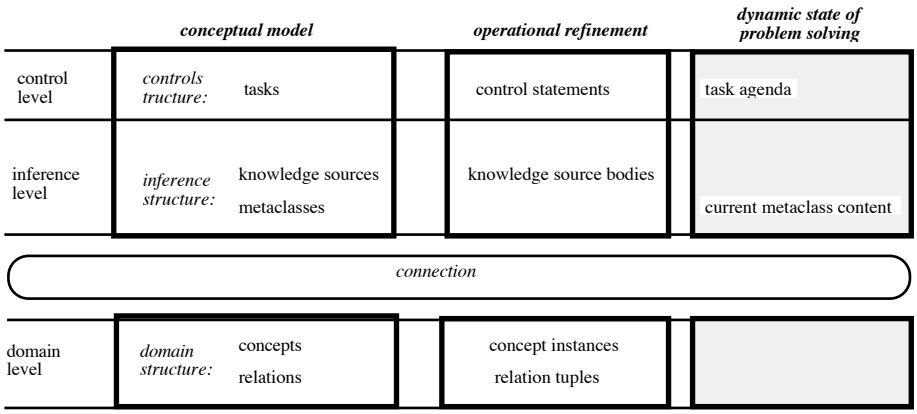

Fig. 1. The MODEL-K framework.

MODEL-K has strongly been influenced by the conceptual models of the KADS knowledge acquisition methodology [4]. In contrast to KADS, MODEL-K separates the definition of concepts and their specific instances, of relations and their tuples, and of tasks and their control statements. KADS models neither have a precise syntax nor are they executable at all. Consequently, any dynamic aspects of problem solving are ignored either. Like KADS, MODEL-K supports the exchange of individual layers. In particular, splitting off the domain layer yields a generic model or system, respectively. In the Esprit Basic Research Action REFLECT [27] [1] we extended both, KADS models of expertise and MODEL-K to incorporate strategic reasoning at a meta-level. This will be the topic of the second part of this article.

MODEL-K is implemented on top of BABYLON, GMD's hybrid knowledge representation workbench which provides – beside prolog and lisp – rule, frame, and constraint formalisms for the definition of the operational parts [8].

2.1 The office allocation application

Planning the arrangement of employees on a floor is a time-consuming process, especially if personnel movement is high as in research institutes. To reach a

satisfactory solution all criteria for a fertile working climate should be considered, e.g. dense communications between projects, proximity of central services, equipment requirements and personal characteristics like smoker aversion.

Given a floorplan with a partial assignment of possible rooms to some employees, and a set of requirements, the office allocation task consists of assigning rooms to all employees satisfying all requirements.

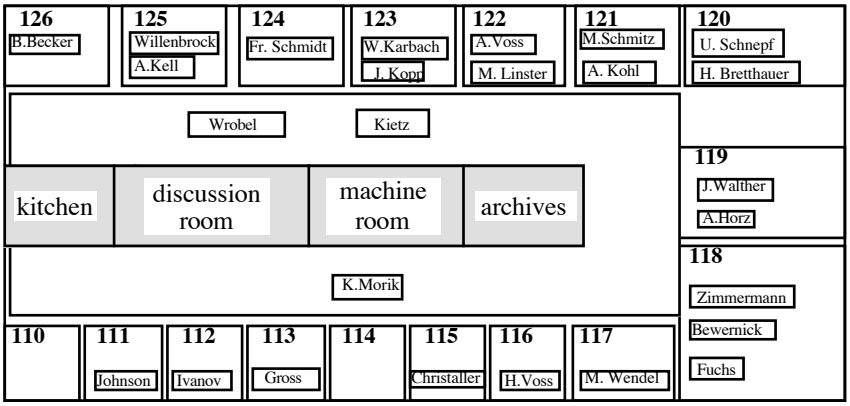

Fig. 2. An office allocation problem.

We treat the problem as an assignment problem where a set of objects, the employees, have to be assigned to another set, the rooms, so as to satisfy the given requirements. We successively match requirements with employees obtaining a constraint network with employees as variables and possible rooms as their values. Global propagation followed by a filtering yields all possible solutions, in case the problem is not overspecified.

The conceptual model of the office allocation problem can be transferred to other domains. For example, in hotel room reservation, the components are the guests and the slots are hotel rooms, in hospital bed allocation the components are the patients and the slots are beds, and in school time table construction, the components are the instruction units and the slots are time intervals.

2.2 The conceptual model

The control layer

At the control layer the flow of control between the knowledge sources is defined in terms of tasks. Tasks may be decomposed into subtasks and knowledge sources. For documentation purposes, a precondition and a goal may be specified. The main task of OFFICE-PLAN consists of an initialize step to select the

components and slots, a step to generate conditions from the generic requirements and an subtask to solve the conditions. Figure 3 shows the entire task decomposition tree.

```
(TASK office-plan
  WITH PRECONDITION = true
       GOAL         = find-an-arrangement
       SUBTASKS     = (initialize integrate-component))
```

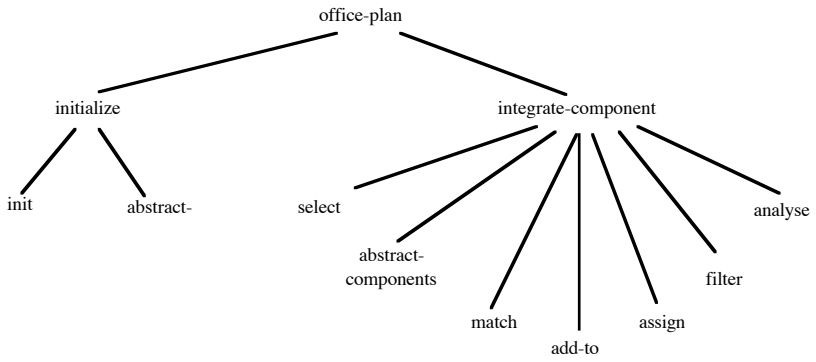

Fig. 3. Task decomposition of OFFICE-PLAN.

The inference layer

At the inference layer, basic inference steps are specified in terms of knowledge sources operating on metaclasses. Both together constitute a graph, the so-called inference structure. Figure 4 shows the one for OFFICE-PLAN.

metaclasses describe the roles domain concepts may play during problem solving. In the spirit of KADS, they achieve an abstraction from any specific domain-layer. Thus, we speak of components instead of employees, and of assignment slots instead of rooms. In MODEL-K, a metaclass can be a structure composed of domain concepts. "set" is a predefined structure. To define others like lists, multisets, stacks, queues or trees might is up to the user.

As an example, metaclass `components-to-place` is initially empty, but will later contain the set of employees that have to be arranged. In other domains, it might contain guests of hotel rooms, airplanes to gates, or of patients of hospital beds. The actual relation between a metaclass and possible domain entries is handled in the connection between inference and domain layer.

```
(METACLASS components-to-place-next-&-their-given-slots
  WITH STRUCTURE         = set)
```

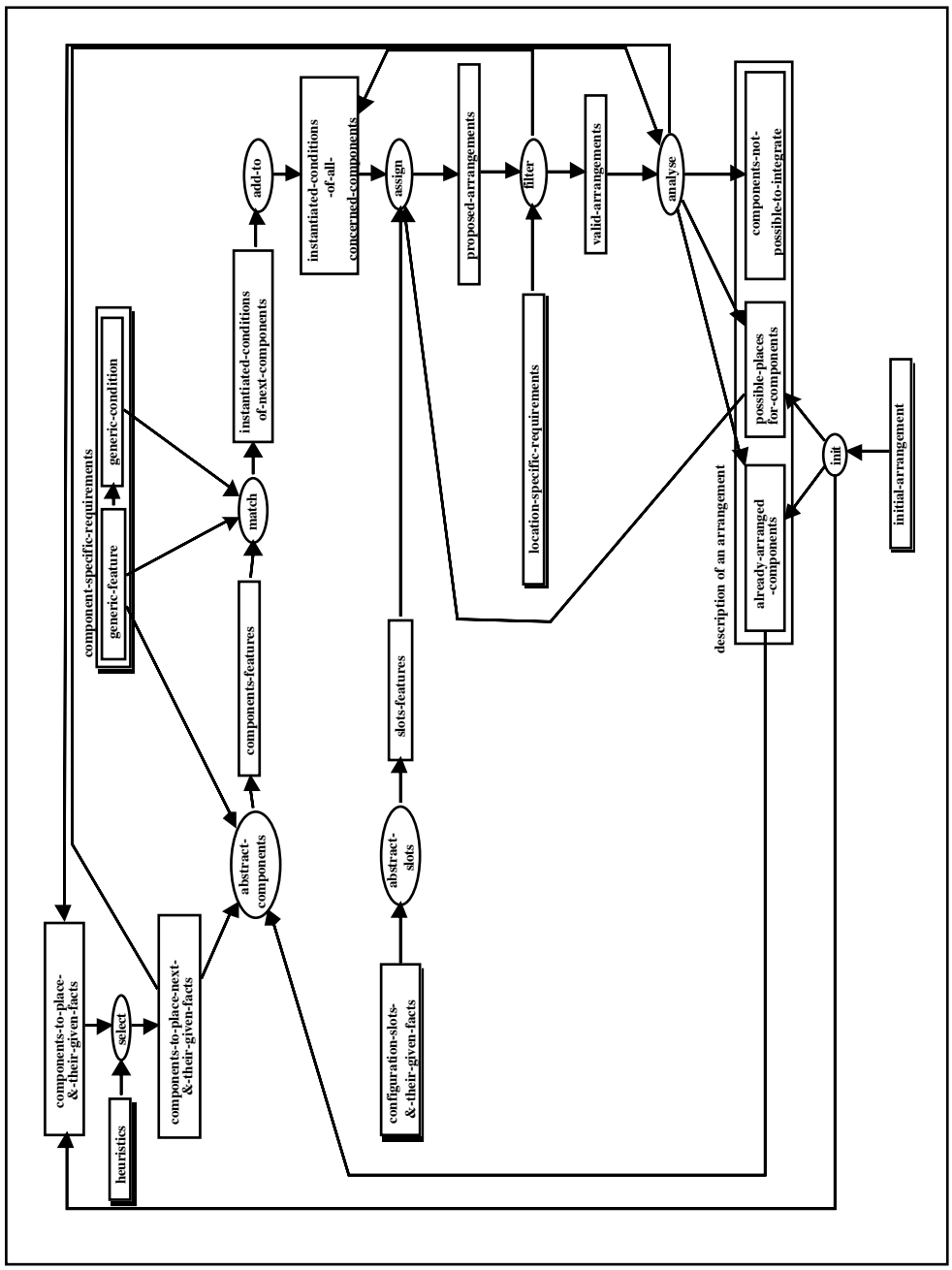

Fig. 4. The inference structure of OFFICE-PLAN.

Knowledge sources describe basic inferences or operations between metaclasses. They are split into a declaration and a definition part. The former specifies input and output metaclasses and the formal relations used in the body. The formal relations keep the knowledge source domain-independent and will be connected to domain-specific relations by separate connection statements. The definition of a knowledge source, usually being quite implementation-biased, is deferred to the operational refinement stage. For instance, knowledge source `abstract-components` will generate component features given certain requirements without needing any domain relations.

```
(KNOWLEDGE-SOURCE abstract-components
  WITH
  DOCUMENTATION-STRING = ...
  INPUT-METACLASSES    = ((components-to-place-next-&-their-given-slots
                           already-arranged-components
                           component-specific-requirements))
  OUTPUT-METACLASSES   = (components-features))
  REQUIRED-RELATIONS   = nil)
```

In OFFICE-PLAN, there are knowledge sources for `init`ialization; to `select` the components to be integrated first according to the given heuristic; to `abstract` essential component and slot features; to `match` the features against the preconditions of requirements in order to instantiate their conditions to constraints; to `add` the newly generated conditions to the constraint network computed formerly; to find all possible `assign`ments of components to slots that satisfy the instantiated conditions; to `filter` them by the slot-specific requirements; and to `analyze` the result for proper termination respectively for the next iteration.

The domain connection

One of the aims of KADS is to develop generic inference and task layers which can be connected to different domain layers. MODEL-K provides language constructs for this connection. For example, the metaclass `components-to-place` is connected to instances of the concept `employee`.

```
(CONNECT-MC components-to-place-next-&-their-given-slots
   WITH DOMAIN-OBJECTS = ((employee))
        UPDATE-UP      = static)
```

To allow the synchronization between the contents of a metaclass and the related objects on the domain layer update procedures can be applied at specific points in time. For example, the metaclasses in a system monitoring a chemical plant are kept consistent with the sensor data received at the domain layer via dynamic updating. For office allocation a static connection upwards for initialization is sufficient.

The formal relations of the knowledge sources are always statically connected to domain-specific relations by:

```
(CONNECT-KS-RELATION <formal-relation> WITH
  DOMAIN-RELATION =  <domain-relation>)
```

The domain layer

The domain layer specifies the ontology of the domain. In MODEL-K, the domain-specific information is stated in terms of predefined basic types like string or integer, user-defined enumeration types, concepts and instances, generic and concrete relations. Together, they constitute the domain structure.

Enumeration Types are similar to the scalar types of procedural programming languages. They define discrete sets with a linear ordering.

In OFFICE-PLAN, we use them to specify the `roles`, `educations`, `groups`, `projects`, `resources` and `themes`. For example, the resources of computing equipment are defined by:

```
(ENUMERATION resources
   WITH values = ((sun macintosh qume symbolics siemens pc)))
```

Concepts are used to represent domain knowledge in an object-oriented way. They may form inheritance hierarchies and describe structural characteristics of domain objects by typed attributes (or slots). In order to suspend the closed-world assumption, the values of a so-called ASSUMABLE attribute may explicitly be specified as to hold, not to hold or to be unknown (see section 2.3 for further details).

In OFFICE-PLAN, `employees`, `rooms`, `requirements`, and `conditions` are defined as concepts. For example, an `office-room` is described by its room number, the number of square meters, the research group it belongs to, and the resources it provides. The default value "-" for each slot means that its value is so far unknown.

```
(CONCEPT office-room
  (SLOTS
 (available-resources - :POSSIBLE-VALUES (:SOME-OF-ENUMERATION resources))
 (belongs-to          - :POSSIBLE-VALUES (:ONE-OF-ENUMERATION groups))
 (room-number         - :POSSIBLE-VALUES :NUMBER)
 (size-of-room        - :POSSIBLE-VALUES :NUMBER)
  (ASSUMABLE
 (available-resources size-of-room) :POSSIBLE-VALUES :ANY)))
```

A requirements is declared by a precondition based on features of employees and a condition:

```
(CONCEPT requirement
  (SLOTS (generic-feature    - :POSSIBLE-VALUES :any)
         (generic-condition  - :POSSIBLE-VALUES
           (:INSTANCE-OF employee-specific-condition))))
```

Relations define dependencies between objects of certain types. In the conceptual model, a relation is declared by its arity, i.e. the types of the components in the tuples, and by properties like transitivity or reflectivity. To suspend the closed-word assumption, ASSUMABLE relations allow to specify for which tuples the relation holds, does not hold, or is unknown (more in section 2.3).

The following example declares a non-assumable, symmetric relation `meeting-often` between two employees:

```
(RELATION meeting-often OF RELATION
  WITH PROPERTIES = ((symmetric))
       TYPE       = ((employee employee))
       ASSUMABLE  = false)
```

Generic relations allow to build classes of relations. For instance, the most important relation in OFFICE-PLAN is `arrangement` which describes arbitrary assignments of employees to rooms. As a concrete instance of this generic relation, `init-arranged` describes which employees are initially placed in which rooms, and which employees are in the hall waiting for an assignment. Other arrangement instances like the proposed solutions will be dynamically constructed at the inference layer.

```
(GENERIC-RELATION arrangement
  (SLOTS (TYPE        (room employee))
         (ASSUMABLE   false)))

(RELATION init-arranged OF arrangement)
```

2.3 The operational refinement

The conceptual model is not yet operational. Concept instances, relation definitions, knowledge source bodies, control statements ordering the subtasks, and connection procedures still have to be supplied. Being mainly a programming task, this is done in the operational refinement.

The control layer

Control statements: In the control structure, each task specifies its constituent knowledge sources and subtasks. Now the flow of control between them is specified by a control statement which may be composed of sequential, branching and loop statements. For example, the `office-plan` task once executes `initialize` and then repeatedly calls `integrate-component` until there are no more components to arrange.

```
(TASK-BODY office-plan ()
    (SUBTASK initialize)
    (WHILE (> 0 (length (<- components-to-arrange :GET-TRUE 'VALUE)) 0)
         DO ((CALL integrate-component))))))
```

Below we show a trace of the main task in OFFICE-PLAN.

```
(<- office-plan :ACTIVATE :TRACE)

--> activating task 'OFFICE-PLAN'
    it's precondition 'TRUE' is satisfied
    --> STATEMENT
        working on (CALL INITIALIZE)
        --> activating task 'INITIALIZE'
            it's precondition 'TRUE' is satisfied
            --> STATEMENT
                working on INIT
                ==> applying knowledge-source 'INIT' ...
                working on ABSTRACT-SLOTS
                ==> applying knowledge-source 'ABSTRACT-SLOTS' ...
            <-- STATEMENT
        <-- ACTIVATE
           ...
```

The inference layer

Knowledge source bodies: The complexity of KADS knowledge sources may vary considerably, from simple access or test functions to complete problem solvers of their own. And often, they cannot be operationalized without taking efficiency into account. This is why we offer the full BABYLON languages for their definition: prolog, rules, constraints, message passing and lisp. As an example, knowledge source `assign` was implemented to call BABYLON's constraint satisfaction interpreter with the network of instantiated conditions:

```
(KNOWLEDGE-SOURCE-BODY assign ()
  (SATISFY current-net :GLOBALLY :WITH current-varibale-domains))
```

The domain connection

MODEL-K provides default connection procedures for metaclasses and formal relations. If no user-defined connection procedure is given, a metaclass is connected statically. By default, a formal relation is connected to its domain relation by matching the arguments position-wise.

```
(CONNECT-KS-RELATION-BODY <formal-task-name> ()
  <body>)

(CONNECT-MC-UP-BODY <metaclass> ()
  <body>)

(CONNECT-MC-DOWN-BODY <metaclass> ()
  <body>)
```

The domain layer

Instances of concepts describe concrete objects in the domain. They inherit all slots and default values from their concept. For instance, `C5-121` is a particular office room. It belongs to group xps, has a number, and has 10 square meters. There are two Macintosh (for simplicity, we did not distinguish between individual machines) and a Sun computer, but definitely no qume terminal. But it is unknown whether there are any siemens terminals, symbolics machines, or pcs.

```
(CONCEPT-INSTANCE C5-121
   OF office-room
   WITH available-resources = (((:TRUE macintosh macintosh sun)
                               (:FALSE qume)))
        belongs-to = xps
        room-number = 121
        size-of-room = 10)
```

The employee-specific requirements in OFFICE-PLAN are: "The head of group and the secretary should be in next-door rooms." "The head of group and each head of project should be in near rooms." "A smoker and a non-smoker should be in different rooms." "Two persons meeting often should be in different rooms." "Persons with no common themes should be in different rooms." For example, the smoker requirement is modelled by:

```
(CONCEPT-INSTANCE smoker-and-not-smoker-aversion-respected
  OF employee-specific-requirement
  WITH generic-feature = ((smoker-and-not-smoker-pair _em1 _em2))
       generic-condition = should-be-in-different-rooms)
```

Our resource-specific requirements are: "A room should provide enough place to its inhabitants." "A room should provide enough machines to its inhabitants." "Nobody should sit together with more fellow-lodgers than he can bear." For example, the requirement providing each employee with a large enough room is defined by:

```
(CONCEPT-INSTANCE provide-space
  OF resource-specific-requirement
  WITH
  generic-feature = ((needs-space _empl _num))
  generic-condition = should-provide-space)
```

Relations are sets of tuples, which can be defined extensionally by enumeration or intensionally by a characteristic predicate. Both alternatives promote prolog as a suitable definition language. Since MODEL-K is implemented in BABYLON, we use BABYLON's special list-notation for Horn-clauses (c.f. [8, p. 147ff]). For instance, the relation `meeting-often` is defined as follows:

```
(RELATION-BODY meeting-often
  ((meeting-often _em1 _em2) <- (close-friends _em1 _em2))
  ((meeting-often _em1 _em2) <- (are-in-same-projects _em1 _em2))
  (meeting-often john-maier john-mayer))
```

Specifying missing knowledge: As in the example of room C5-121, knowledge may sometimes be incomplete. We definitely know that there are two Macintosh and one Sun in the room. But, under the closed-world assumption that holds for ordinary slots and relations, we cannot express that there definitely is no qume terminal while we do not know anything about the other machines. For such situations, MODEL-K offers the ASSUMABLE attributes and relations. They allow to specify which objects are attribute values or relation tuples, which are not, and which are unknown.

Assumable slots are usually multi-valued. They are interpreted as partial multisets, i.e. as partially defined functions from their POSSIBLE-VALUES domain to the natural numbers. In the example of room C5-121, macintosh is mapped to 2, sun to 1 and qume to 0. To ease the definition of assumable attributes we added abbreviations for cases without unknown values. Moreover, the set of unknown values can always be deduced and need not be specified. To access and modify the values of assumable attributes under a specific truth modality MODEL-K provides predefined methods. For example, after sending the message `(<- C5-121 :add-true 'available-resources '(symbolics))` there will also be a Symbolics in the room. The consistence of the partial multiset is automatically maintained.

Assumable relations are implemented by extending the tuples by a boolean argument. When adding a tuple its status has to be specified in the last component. The example shows the tuples of an assumable relation `close-friends`.

```
(RELATION-BODY
 ((close-friends _em1 _em2 _truth) <- (married _em1 _em2 _truth))
 ((close-friends john-meier john-meyer false)))
```

Again, there are predefined methods to get all true, false or unknown tuples. However, MODEL-K does not provide an interpreter for reasoning under incomplete knowledge so that the knowledge engineer himself must implement the correct inference procedures as was done in the first clause of the example to propagate unknown knowledge.

Switching representation: Since logically the slots of an instance can be viewed as binary relations, MODEL-K provides a uniform PROVE method for both, concept attributes and relations, easing the transition between both representations during development.

For example, `(PROVE '(initial-arrangement _X 'C5-121))` finds all persons being seated in room C5-121. And `(PROVE '(available-resources _X 'sun))` finds all office-room instances with a sun.

This gives the knowledge engineer the freedom to switch representation formalisms during development without having to change other parts of the description.

2.4 Supporting prototyping

Usually, knowledge sources are called from the control layer. For prototyping, i.e. testing knowledge sources or simulating a not yet existing control layer, it is useful to activate individual knowledge sources by hand. For that purpose, the interpreter can execute (SUBTASK $<$ *task* $>$) and (KS $<$ *knowledge* $-$ *source* $>$) statements for activating tasks and knowledge sources, respectively. For accessing and modifying the status of metaclasses during the development process, a uniform protocol in terms of :GET, :RESET, :ADD and :REMOVE methods is provided to inspect, reset and modify metaclasses.

3 The strategy layer: Meta-reasoning in MODEL-K

The strategy layer in KADS was only vaguely described and has hardly been used in any KADS model. While strategic reasoning is usually concerned with control issues [14], we would like to incorporate some more capabilities at the strategic layer: checking the solvability and difficulty of a problem, predicting and scheduling the resources for problem solving, problem simplification, detecting deadends, and repairing impasses. All these tasks require reasoning *about* the underlying system and thus are meta-activities.

Although the domain of these meta-activities is another knowledge-based system, they can be described in terms of generic problem solving methods like diagnosis, assessment or repair. That means the strategy layer is regarded as another problem solver at the meta-level. Therefore, we can use the same modelling scheme as for the lower three layers, namely another control, inference, and domain layer. Control and inference layers describe the problem solving method to be carried out at the meta-level, for example diagnosing the feasibility of a problem. The domain layer contains the lower three layers to be reasoned about and often special meta-knowledge, like the consistency of data in the object system or necessary conditions to solve a problem.

Since the meta-system reasons about the lower three layers while they are "running", these layers and the object system they model must be kept consistent. Any progress in the object system must be reflected upwards into the model, and any modifications of the model must be propagated downwards into the system. In the terminology of Maes, we need a causal connection, and the object and meta-system together constitute a reflective system [19].

In MODEL-K, we need not worry about the causal connection. Since a system is just an extension of its conceptual model, every object system incorporates its model. You simply cannot change one without the other.

As it turns out, we now have two systems: A meta-system implementing the strategy layer and an object system representing the lower control, inference and domain layers (c.f. figure 5) . This separation has the advantage that meta-systems (and their conceptual models) can be kept generic and reusable for other object systems. For example, predicting the complexity of a problem by estimating the search space is applicable to a wide range of object systems.

In the REFLECT project, we focused on competence assessment, improvement, and resource management. We came up with a range of generic meta-behaviors, most of which we applied to the conceptual model of our office room allocation system. Examples are feasibility studies to detect overcomplex or over-specified problems, relaxing inconsistencies, removing redundancies, controlling time restrictions, etc. [35] [36].

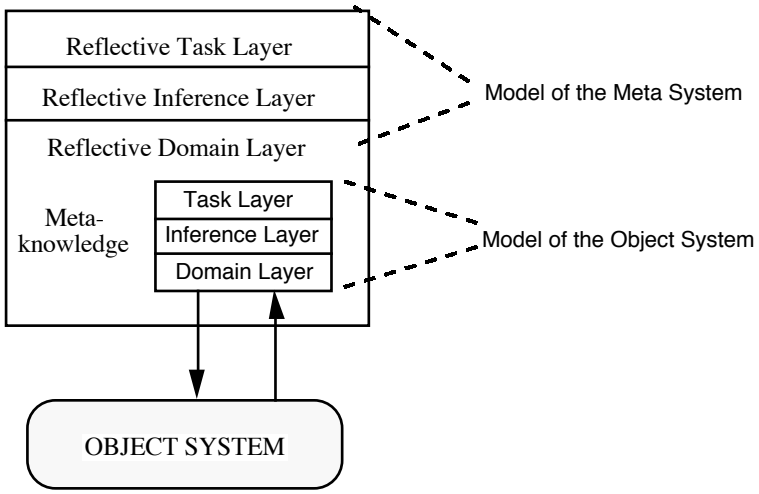

Fig. 5. Knowledge level view of reflective systems.

3.1 Design decisions

Though the idea to use MODEL-K for both, object system and meta-system, seems to be straightforward, a few design decision had to be made which are left open in the conceptual framework. One concerns consistency and the other the control problem of switching between the two systems.

Consistency between the object system and its model

While solving a problem, the object system changes dynamically. For reasons of consistency, these changes must be "reflected" upwards into the model. Vice

versa, reflective problem solving operates on its model of the object system, assuming that any changes are "reflected" downwards. (Reflection upwards and downwards are terms introduced by [39] in the context of the logical language FOL). As an example, our time management system repeatedly invokes the assign and filter tasks of OFFICE-PLAN. Based on the number of assignments proposed or filtered, it then decides which task to call next for which time slice. Thus, the number of assignments proposed or filtered has to be repeatedly passed upwards, while the time slices must be passed downwards.

However, such a causal connection is only required if the object model is separated from the object system, which is not the case with MODEL-K. Instead we encounter a similar synchronization problem between the object model and the meta-inference layer. That is because the metaclasses here may contain information from the object model which dynamically changes with the object system. Therefore, the metaclasses at the meta-level must be kept consistent with the object model. An example are the metaclasses for time management which contain the time slices resp. the numbers of solutions. Vice versa, information from meta-level metaclasses may have to be passed downwards into the object model (and thus automatically into the object system).

To maintain consistency, we have three alternatives for both directions. We can lazily update the object system, respectively its model, just before any data item is accessed, we can eagerly propagate all changes directly, or we can propagate them before switching control between the two systems. For MODEL-K we decided to propagate both the changes of the object system to the meta-level metaclasses and vice versa at switching time as a default, reducing the number of update operations. For additional intermediate updates, explicit update statements are provided at the meta-level task layer.

Independently of when we synchronize, we may either update the object system respectively its model completely or incrementally. The former means more copy operations, while the latter requires keeping track of what has changed. For MODEL-K we decided to update incrementally, namely upwards everything that is referenced in a meta-level input metaclass and downwards every output metaclass.

Switching paradigm

Since we have two systems, we have to decide which one should be active and when it should pass over control. For example, for time management we must somewhere state that office planning must proceed up to the assign step. Similarly, the time management system must first perform certain analyses before it can start controlling the assign-filter phase. Afterwards, both systems must be run to completion. In [32] and [27], different switching paradigms were suggested:

Meta-simulation: All control is with the meta-system. Whenever necessary, it has the means to simulate the object system. This alternative requires more knowledge about the object system than we have at the knowledge level.

Asynchronous communication: Meta- and object system run in parallel. Thus it may happen that the conclusions of the meta-part have become obsolete or no longer apply because of the progress in the object system.

Crisis management: The object system operates until it recognizes a crisis. It then passes control to the meta-part, which may modify the object system and then reactivate it. One problem with this paradigm is that the object system must be able to recognize its own crises and deadends. Another problem is, that the meta-system has no chance to prevent any crises.

Reflect-and-act: After each elementary inference step, control is passed from the object system to the meta-system. There the object system may be modified before control is returned. In this paradigm, the elementary inference steps chosen define the grain size for reflection. In the KADS framework, the knowledge sources are natural candidates, since we cannot inspect them any deeper at the knowledge level. Control in the meta-system is necessarily event-driven, which does not readily match with the KADS task structure. As another disadvantage, control is switched more often than necessary.

Subtask management: The meta-system may activate its own tasks as well as tasks in the object system. This paradigm nicely fits into the KADS framework. A meta-task, instead of calling a meta-subtask, would be allowed to call a task in the object system, which could then invoke further object level subtasks. The problem with this alternative is, that there is only one meta-system having exclusive control about the object system.

Instead of having one large meta-system we want to be able to develop small specialists on top of the object system, each tackling an elementary meta-task, like a feasibility expert, a relaxation expert, or a redundancies expert. They should be collected in a library. Given a particular object system, we would like to choose some meta-modules from the library and combine them. The combination should involve minimal changes with the individual meta-systems only.

External switching In order to overcome this problem with subtask management, we came up with a new alternative. We introduced an external scheduler that controls the switches both between the individual meta-specialists and between them and the object system. That means it can call tasks that are defined either at some meta-task layer or in the object system. The task layers of the meta-specialists only contain local tasks. They need not be modified when being combined with other meta-systems. Only the individual scheduling layers must be replaced by a joint one.

Figure 6 shows the resulting scheme of extended MODEL-K. There is one object system with only one model that is accessed from all meta-components. Such a component may have additional meta-domain knowledge, inference and task layers. The individual schedulers are replaced by one common scheduling layer. Examples of meta-domain knowledge may be the pairs of redundant or contradictory constraints, or knowledge about the importance of requirements, constraints, or constraint variables like the employees in office planning.

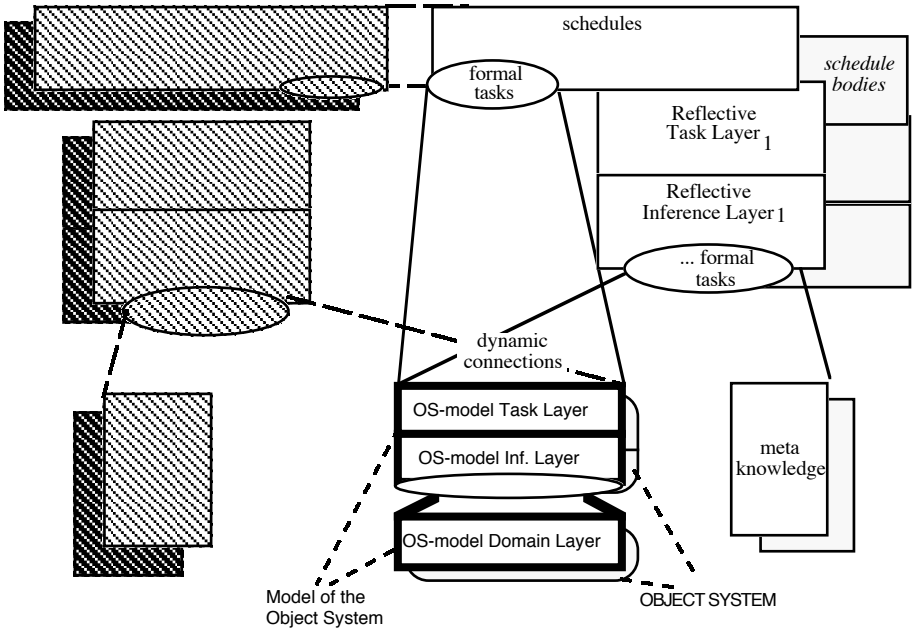

Fig. 6. Schema of extended MODEL-K

Since the scheduler should be generic, it must not directly refer to tasks of the object system. For that reason we introduced formal task names that can be connected to concrete ones when the scheduler is connected to a concrete object system. These formal task names are comparable to the formal relation names we already need in ordinary models. They are used by knowledge sources at the inference layer to generically refer to relations defined at a concrete domain layer.

Calling object system tasks from the scheduler is not always sufficient. Sometimes a meta-knowledge source needs to control certain tasks in the object system exclusively. For instance, the time management system has an inference step invoking the assign and filter steps of the object system with fixed time slices. As for the scheduler, these references are generically made via formal task names.

3.2 Example: Predicting deadends in problem solving

We will motivate why contradictory conditions in office-planning problems can occur at all and how they are removed by the meta-module `CONTRA-C`.

We will illustrate MODEL-K's meta-approach to strategic reasoning by CONTRA-C, a small competence specialist for OFFICE-PLAN. It prevents the object system from running into a deadend by checking the solvability of

a problem in advance and proposing amendments to make the problem solvable. As already said, OFFICE-PLAN transforms requirements, like separate-smoker-and non-smokers into a constraint network. Assume we obtain the following constraints: `same-rooms (Monika, Uli), next-door-rooms (Thomas, Monika), near-rooms(Thomas, Hans),`
`different-rooms (Monika, Uli), ...` Obviously[2] no consistent assignment can be found because the constraints between Monika and Uli are contradictory.

As shown in figure 7, the task of the meta-module CONTRA-C is decomposed into subtasks for diagnosing (i.e. detecting) and repairing (i.e. removing) contradictory conditions. Diagnosis consists of analyzing the object system and interpreting the obtained findings according to fault categories. Repair consists of proposing repairs and applying them.

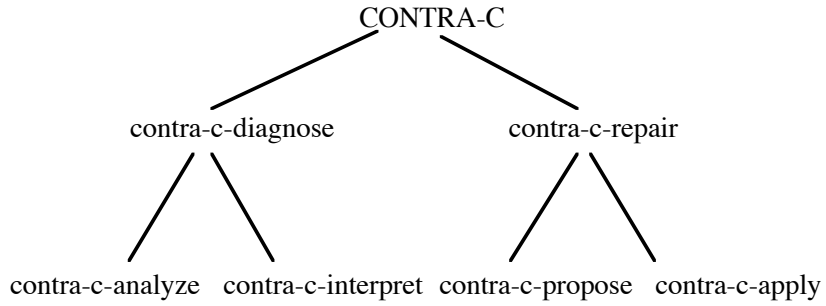

Fig. 7. Task structure of the meta-module

Figure 8 shows the inference structure of CONTRA-C. metaclass `OS-component-conditions` is filled with the relevant information from the object model, which is the constraint network. It is inspected by knowledge source `contra-c-analyze` in order to detect pairs of contradictory conditions. If contradictions have been found knowledge source `contra-c--interpret` adds the fact `(inconsistent-knowledge true)` to metaclass `ml-malfunctions`. `contra-c-propose` then asks the user to remove one condition from each pair of contradiction. The result is passed via metaclass `ml-repairs-remove-contra-c-result` to `contra-c-apply` which removes the chosen conditions from the network.

At its domain layer CONTRA-C needs knowledge about how to detect pairwise contradictory constraints, in its simplest form an opposite relation between pairs of constraints.

We used MODEL-K to model and implement ten competence specialists incorporating strategic knowledge to assess and improve the competence of OFFICE-PLAN: FEASI detects unsolvable problems by comparing available and

[2] Although this seems easy to detect for us, we should not forget that the words 'different' and 'same' have no meaning for the object problem solver.

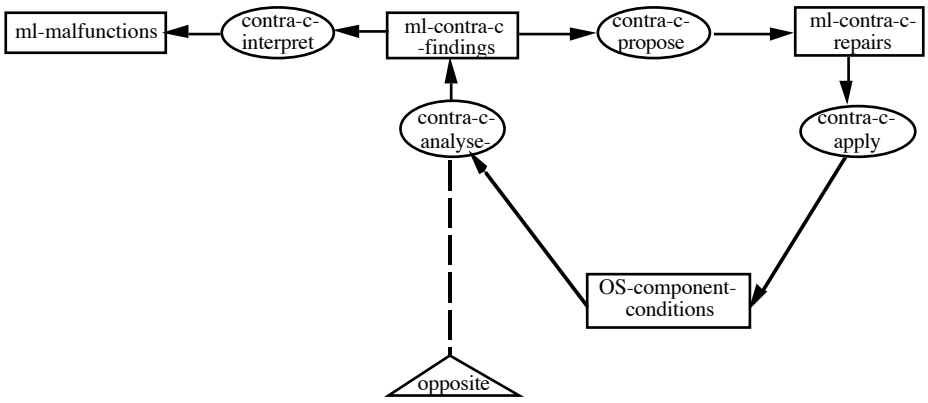

Fig. 8. Inference structure of the meta-module

required resources. CONTRA-R and CONTRA-C detect and remove contradictory requirements from the original problem statement resp. contradictory constraints from the internal representation. RED-R, RED-C, RED-CF, and SIMPLE-D simplify the problem by removing redundancies between generic requirements, between constraints, and by reducing the possible values of constrained variables. COMIC decomposes overcomplex problems and solves them while reusing previously found partial solutions. It can return approximate solutions by suitably composing solved subproblems. TACKLE-TIME allows to limit the time and the number of solutions of the object system by scheduling time slices between a generate and a test subtask (in our case propose and evaluate). One of our partners, BSR-Consulting, built CASY which uses a library of complete cases to shortcut the entire problem solving process.

3.3 Related work

Although our work concerning the strategic layer was influenced by the work on computational reflection [19] [29], it differs in the kind of model used to reflect upon. Whereas languages like KRS or 3-Lisp allow to access the implementation and the run-time environment of programs like class-methods or the interpreter stack, we stress an abstract, implementation-independent model of the underlying object system. We also abandoned the idea of infinite meta-level towers generated by meta-circular interpreters, as we do not see how to acquire knowledge for a third or any higher levels. Work on reflection in logic has shown that a clear separation between object- and meta-level ensures avoidance of paradoxes [24].

ML^2 [33] is a language to formalize KADS knowledge level models. It has been used to explicitly represent the model to be reasoned about. Although ML^2 is in logic and hence more declarative than MODEL-K, the language lacks operational support for the causal connection and the integration of different

reflective modules.

The meta-modules developed in MODEL-K are in line with ideas about meta-level reasoning presented in [10], [15], or [37], and recently published work on reasoning under resource limitations [26].

4 Summary

MODEL-K is a language for describing KADS knowledge level models as well as their operational refinements. Since the resulting systems incorporate their conceptual model, this language supports prototyping almost at the knowledge level. Being so close to the knowledge level, the systems are easier to understand by experts and users, they are easier to extend, maintain and explain. But not only knowledge engineers and users profit from this transparency. Knowledge based meta-systems will monitor, assess and improve object systems automatically. The built-in conceptual model in MODEL-K defines the interface for the meta-layer and provides access to the underlying system. Since the meta-system is regarded as another problem solver, the MODEL-K modeling framework can be applied to both, object- and meta-system.

Just like KADS strives at building a large library of knowledge level models, we would like to collect corresponding operationalizations in MODEL-K. Other than generic tasks [6] or the systems of McDermott's group [20], the MODEL-K systems should be easy to adapt and to combine, being written in the same high level language. We are looking forward to results of the KADS II Esprit project, where executable methods shall be developed for the models in the KADS library. Like for object systems, we are aiming at a library of generic meta-systems which can be supplied to classes of object systems.

With its interpretation of the strategic layer, MODEL-K provides a modeling and implementation framework for meta-reasoning systems. To flesh it, more meta-level knowledge and theories have to be acquired. Now the experts are asked –programmers and knowledge engineers– to provide their knowledge *about* how to assess and improve problem solvers.

Acknowledgements

We thank Brigitte Bartsch-Spörl and Hans Voß for comments on earlier versions of this paper, and Ralph Schukey and Uwe Drouven for their cooperation in designing and implementing MODEL-K.

Appendix: The constructs of reflective MODEL-K

insert table with MODEL-K constructs here

References

1. B. Bartsch-Spoerl, B. Bredeweg, C. Coulon, U. Drouven, F. van Harmelen, W. Karbach, M. Reinders, E. Vinkhuyzen, and A. Voß. Studies and experiments with reflective problem solvers. ESPRIT Basic Research Action P3178 REFLECT, Report IR.3.1,2 RFL/BSR-UvA/II.2/1, REFLECT Consortium, August 1991.
2. B. Bartsch-Sporl, M. Reinders, H. Akkermans, B. Bredeweg, T. Christaller, U. Drouven, F. van Harmelen, W. Karbach, G. Schreiber, A. Voß, and B. Wielinga. A tentative framework for knowledge-level reflection. ESPRIT Basic Research Action P3178 REFLECT, Deliverable IR.2 RFL/BSR-ECN/I.3/1, BSR Consulting and Netherlands Energy Research Foundation ECN, August 1990.
3. C. Beierle, W. Olthoff, and A. Voß. Towards a formalization of the software development process. In *Software engineering 86*, IEE Computing Series 6, London, 1986. PeterPeregrinus Ltd.
4. J. A. Breuker and B. J. Wielinga. Model Driven Knowledge Acquisition. In P. Guida and G. Tasso, editors, *Topics in the Design of Expert Systems*, pages 265–296, Amsterdam, 1989. North Holland.
5. B. Chandrasekaran. Generic tasks in knowledge-based reasoning: High-level building blocks for expert system design. *IEEE Expert*, 1(4):279 – 299, 1986.
6. B. Chandrasekaran. Generic tasks as building blocks for knowledge-based systems: The diagnosis and routine design examples. *The Knowledge Engineering Review*, 3(3):183–210, 1988.
7. D.C. Brownand B. Chandrasekaran. *Design Problem Solving: Knowledge Structures and Control Strategies*. Research Notes in Artificial Intelligence. Pitman, London, 1989.
8. Th. Christaller, F. di Primio, and A. Voß. *The AI Workbench BABYLON: An Open and Portable Development Environment for Expert Systems*. Academic Press, London, 1992.
9. W.J. Clancey. From Guidon to Neomycin and Heracles in twenty short lessons. *The AI Magazine*, 7(3):40–61, August 1986.
10. R. Davis. Applications of meta-knowledge to the construction, maintenance, and use of large knowledge-bases. AI memo 283, Stanford University, Palo Alto, July 1976.
11. R. Davis and B. G. Buchanan. Meta-level knowledge: Overview and applications. In *IJCAI-77*, pages 920 – 927, Cambridge MA, August 1977.
12. D. Fensel, J. Angele, and D. Landes. KARL:: A knowledge acquisition and representation language. In J.C. Rault, editor, *Proceedings of the 11th International Conference Expert systems and their applications*, volume 1 (Tools, Techniques & Methods), pages 513 – 528, Avignon, 1991. EC2.
13. Ch. Floyd. A systematc look at prototyping. In R. Budde, K. Kuhlenkamp, L. Mathiassen, and H. Züllighoven, editors, *Approaches to Prototyping*, pages 1–18. Elsevier, Berlin, 1984.
14. T. R. Gruber. *The acquisition of strategic knowledge*, volume 4 of *Perspectives in artificial intelligence*. Academic Press, Boston, 1989.
15. E. Hudlicka and V.R. Lesser. Meta-level control through fault detection and diagnosis. In *Proceedings of the National Conference on Artificial Intelligence*, pages 153 – 161, Austin, Texas, 1984.
16. W. Jansweijer. *PDP*. PhD thesis, University of Amsterdam, 1988.
17. W. Karbach, M. Linster, and A. Voß. Models, methods, roles and tasks: many labels - one idea? *Knowledge Acquisition journal*, 2:279 – 299, 1990.

18. Marc Linster. Linking modeling to make sense and modeling to implement systems in an operational environment. In Thomas Wetter, Klaus-Dieter Althoff, John Boose, Brian Gaines, Marc Linster, and Franz Schmalhofer, editors, *Current developments in knowledge acquisition: EKAW92*, LNAI, Heidelberg, 1992. Springer.
19. P. Maes. Computational reflection. Technical report 87-2, Free University of Brussels, AI Lab, 1987.
20. S. Marcus, editor. *Automatic knowledge acquisition for expert systems.* Kluwer, 1988.
21. M.A. Musen. *Automated Generation of Model-Based Knowledge-Acquisition Tools.* Pitman, London, 1989. Research Notes in Artificial Intelligence.
22. R. Neches, W. Swartout, and J. Moore. Explainable (and maintainable) expert systems. In *IJCAI-85*, Los Angeles, 1985.
23. A. Newell. The knowledge level. *Artificial Intelligence*, 1982:82–127, 1982.
24. D. Perlis. Languages with self-reference I: Foundations. *Artificial Intelligence*, 25:301–322, 1985.
25. F. Puppe. Med2: How domain characteristics induce expert system features. In H. Stoyan, editor, *GWAI-85*, pages 272–284. Springer-Verlag, 1986.
26. S.J. Russell and S. Zilberstein. Composing real-time systems. In *Proceedings of the 12th International Joint Conference on Artificial Intelligence, Sydney, Australia*, volume 1, pages 212 – 217, San Mateo, 1991. Morgan Kaufmann.
27. G. Schreiber, B. Bartsch-Sporl, B. Bredeweg, F. van Harmelen, W. Karbach, M. Reinders, E. Vinkhuyzen, and A. Voß. Designing architectures for knowledge-level reflection. ESPRIT Basic Research Action P3178 REFLECT, Deliverable IR.4 RFL/UvA/III.1/4, REFLECT Consortium, August 1991.
28. H. E. Shrobe. Dependency directed reasoning in the analysis of programs which modify complex data structures. In *IJCAI-79*, pages 829–835, Tokio, 1979.
29. B. Smith. Reflection and semantics in a procedural language. Technical Report TR-272, MIT, Computer Science Lab., Cambridge, Massachussetts, 1982. Also in: *Readings in Knowledge Representation*, Brachman, R.J. and Levesque, H.J. (eds.), Morgan Kaufman, California, 1985, pp. 31-40.
30. M. Stefik. Planning and meta-planning (molgen: Part 2). *AI journal*, 16:141 – 170, 1981.
31. G. J. Sussman. *A Computer Model of Skill Acquisition*, volume 1 of *Artificial Intelligence Series.* American Elsevier, New York, 1975.
32. F. van Harmelen. *Meta-level Inference Systems.* Research Notes in AI. Pitmann, Morgan Kaufmann, London, San Mateo California, 1991.
33. F. van Harmelen and J. Balder. $(ML)^2$: A formal language for kads models of expertise. *Knowledge Acquisition*, 4, Special Issue in KADS(1):127 – 159, 1992.
34. J. Vanwelkenhuysen and P. Rademakers. Mapping knowledge-level analysis onto a computational framework. In L. Aiello, editor, *Proceedings ECAI'90, Stockholm*, pages 681–686, London, 1990. Pitman.
35. A. Voß, W. Karbach, B. Bartsch-Spoerl, and B. Bredeweg. Reflection and competent problem solving. In Th. Christaller, editor, *GWAI-91, 15th German Workshop on Artificial Intelligence*, pages 206 – 215, London, 1991. Springer Verlag.
36. A. Voß, W. Karbach, C.H. Coulon, U. Drouven, and B. Bartsch-Spoerl. Generic specialists in competent behavior. In *Proceedings of ECAI-92*, 1992.
37. A. Voß, W. Karbach, U. Drouven, and D. Lorek. Competence assessment in configuration tasks. In L.C. Aiello, editor, *Proceedings of the 9th European Conference on Artificial Intelligence*, pages 676 – 681, London, 1990. ECCAI, Pitman.
38. T. Wetter. First-order logic foundation of the KADS conceptual model. In B. Wielinga, J. Boose, B. Gaines, G. Schreiber, and M. van Someren, editors, *Cur-*

rent trends in knowledge acquisition, pages 356–375, Amsterdam, May 1990. IOS Press.
39. R. Weyhrauch. Prolegomena to a theory of mechanized formal reasoning. *Artificial Intelligence*, 13, 1980. Also in: *Readings in Artificial Intelligence*, Webber, B.L. and Nilsson, N.J. (eds.), Tioga publishing, Palo Alto, CA, 1981, pp. 173-191. Also in: *Readings in Knowledge Representation*, Brachman, R.J. and Levesque, H.J. (eds.), Morgan Kaufman, California, 1985, pp. 309-328.
40. B. Wielinga and J. Breuker. Models of expertise. In *Proceedings ECAI'86*, pages 306–318, Brighton, 1986.

This article was processed using the LaTeX macro package with LLNCS style