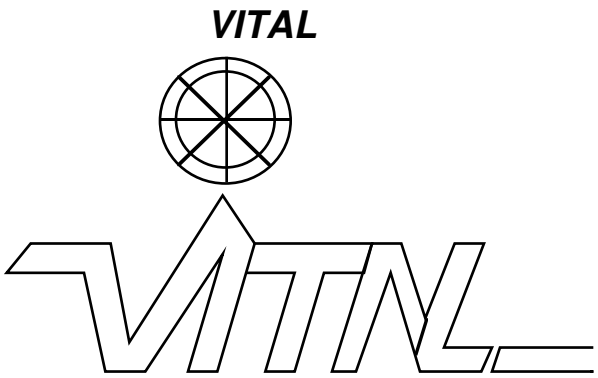
ESPRIT II Project P5365



A methodology - based workbench for KBS Life cycle support

Document reference:	VITAL/ID253.4	Date : 22.xii.94
Activity :	T253.4	Status : DRAFT
Distribution :	T253 partners	
Title :	Reusable and Maintainable KBS Design	
Authors :	Arthur STUTT, Enrico MOTTA	
Abstract :	The VITAL Design Methodology is augmented with a means of recording argumentation so as to produce reusable knowledge level models.	
Keywords :	Design, Argumentation	

Collaborators (VITAL Partners) :

SYSECA - SYSECA TEMPS REEL
BULL - BULL CEDIAG
ONERA - ONERA
OU - THE OPEN UNIVERSITY (*)
NOTT - UNIVERSITY OF NOTTINGHAM
AC - ANDERSEN CONSULTING
PTT - ROYAL PTT NEDERLAND NV
NOKIA - NOKIA RESEARCH CENTER
Partners marked (*) are involved in this task

©1994 The VITAL Consortium

Table of Contents

1. Introduction 1
1.1. The design of KBSs.....2
1.2. The knowledge level.....3
1.3. A knowledge level model of design 3

2. Methodological Background..... 4
2.1. Tasks and problem solving methods 4
2.2. The design of KBS—A survey of Methodologies.....8
2.2.1. How does designing KBSs differ from designing other software?.....8
2.2.2. A framework for discussing methods for KBS design.....8
2.2.3. KADS I 8
2.2.4. CommonKADS/KADS II.....13
2.2.5. Components of Expertise (CoE) 16
Comments.....17
2.3. The VITAL knowledge engineering methodology.....18

3. Design and Argumentation..... 19
3.1. The Design Rationale.....20
3.2. Non-argumentative design.....21
3.3. Models of Argumentation.....21
3.4. Approaches to design and argument.....22

4. Languages for KBS Design 23
4.1. Languages for KBS design 23
4.2. A decision based design language for KBS design.....25
4.3. Design Principles—the form of the library 28
4.4. Design Steps—Operational Design Principles.....31
4.5. OCML-DDL—Mini-Manual for language 32

5. An Example—OCML-DDL in use..... 35

6. Conclusion—OCML-DDL improves the VITAL design methodology 43

References 44

Appendix—The VITAL extended OCML (VITAL-OCML-DDL)..... 46

List of Figures

Figure 1: Knowledge Level Design ……4

Figure 2: A sample task specification: Parametric Design……6

Figure 3: Mapping a role to domain-level knowledge structures ……6

Figure 4: A coarse-grained specification of the Propose&Revise method.....7

Figure 5: The stages in the design process (after Schreiber Figure 6.3) ……10

Figure 6: Application Development in CoE……16

Figure 7: A model of design argumentation ……27

Figure 8: Design as argumentation.....28

Figure 9: The I/O specification for the class of Propose&Revise PSMs ……36

Figure 10: Functional view of the Ontolingua VT KB ……37

Figure 11: The task structure of the Propose&Revise PSM ……37

Figure 12: The possible methods for the Propose task……38

Figure 13: The ProposeAll method is chosen ……40

Figure 14: The task structure of ProposeAll……40

Figure 15: The task structure of Propose1……41

Figure 16: Simple parameter selection method ……41

Table 1: KADS Design Stages and their results (after Tansley and Hayball)……12

Table 2: Summary of KADS-1 design methodology characteristics……13

Table 3: Summary of KADS-II design methodology characteristics ……15

Table 4: Summary of CoE design methodology characteristics ……17

Table 5: Examples of design principles and their indices ……30

1. INTRODUCTION

This report offers a revised version of the VITAL design methodology. This version sees design as taking place at the *knowledge level* as a result of a *process* of *decision-making* (and argumentation). The resulting model contains *reusable* elements and the *rationale* for their use. Section 1 introduces the issues involved in the design of KBS and shows the relevance of the knowledge level. Section 2 outlines the methodological background, providing, in particular, definitions for the terms commonly used in KBS design and a survey of other design methodologies. Section 3 shows how design is intimately related to argumentation. Section 4 reviews some commonly used design languages, provides a process model for design and introduces OCML-DDL—the VITAL decision based design language. Section 5 gives an example (based on the VITAL solution to the SISYPHUS-II Elevator Configuration problem) of OCML-DDL in use while Section 6 makes some tentative conclusions.

It is generally accepted among researchers that knowledge engineering (KE) involves the creation of models. Van de Velde (1994b: 727) calls this the Modeling hypothesis. Since he views models as "the result of a series of arguments, negotiations and circumstances involving knowledge engineers, domain experts, clients, system developers, managers and so forth that need to agree on its acceptability for their different purposes" (ibid.) he proposes the creation of *modeling* libraries which will include modeling *operations*. A modeling operation (728) is "an operation (relation) over models that transforms a generic model into another one". Taken collectively, these operations "realize library support in the construction of a model, i.e., as a store of model construction steps and episodes" (ibid.) This is an attractive view. However, while Van de Velde's modeling operations are capable of capturing the different sorts of design processes which transform one design model into another, they do *not* serve to capture "the series of argumentations among interested parties for which it [the model] serves as a common ground for discussion" (730). We propose therefore that, in order to capture the "best practices on knowledge engineering" of a community (731), Van de Velde's proposed library needs to be complemented by a means of storing the argumentation surrounding designs. To do this we will suggest that work on capturing what has been called the design rationale should be adapted to the design of knowledge based systems (KBSs). This provides the best means for storing "modeling episodes" with the "argumentations that annotate them" (728) since there are several well-tried techniques available—techniques which elevate argumentation from mere annotation to its rightful place (the one Van de Velde seems otherwise to accept) at the centre of the modelling process. This approach has much in common with the work of Landes and Studer (1994).

In this report, then, we focus on the process of constructing reusable *knowledge level* models (*à la* Van De Velde - see section 1.1) by augmenting design models with a means of recording argumentation. In particular we discuss an explicit process model of KBS design at the knowledge level and highlight the kind of design decisions which need to be taken at this level of design - e.g. task decomposition, selection of problem solving methods. Our method includes a set of design principles (our equivalent of Van de Velde's modeling operations), an expressive design language for representing design models, and a *decision description* (or *design rationale*) component (for capturing the KE community's practices). Our design principles are multiply indexed in a library so that principles are available to guide design decisions at various levels ranging from the very general to the specific: e.g. when choosing between design methodologies and when choosing between problem solving methods.

We also present a concrete example of the application of our design methodology which illustrates how our design language can be used to produce a communicable design model comprising an explicit record of the design decisions which have been taken during its construction. We believe that this approach improves the reusability of design models as it makes explicit both how model components have been combined by actual designers (the design rationales) and how they may best be combined (design principles).

1.1. The design of KBSs

The design of knowledge-based systems (KBS) is a special kind of software system design, which, in turn, has much in common with other forms of design. According to Simon a science of design is possible: "a body of intellectually tough, analytic, partly formalizable, partly empirical, teachable doctrine about the design process." (1981: 132).

To comply with this characterisation, the designer of a KBS, like the designers of other types of software, needs: a formal or semi-formal representation (whether graphical or textual) for expressing a design; tools for aiding the process; and, some methodology for creating (including re-use of components) and structuring the design of the target system. In addition there are some elements which are specific to KBS design. These include: the existence of specific techniques (drawn from AI) for operationalising tasks and data storage; the existence of methodological guidelines tailored to the production of KBS designs; and, most importantly, the emphasis placed on the explicit representation and use of knowledge, and on modelling the problem solving behaviour of human experts.

1.2. The knowledge level

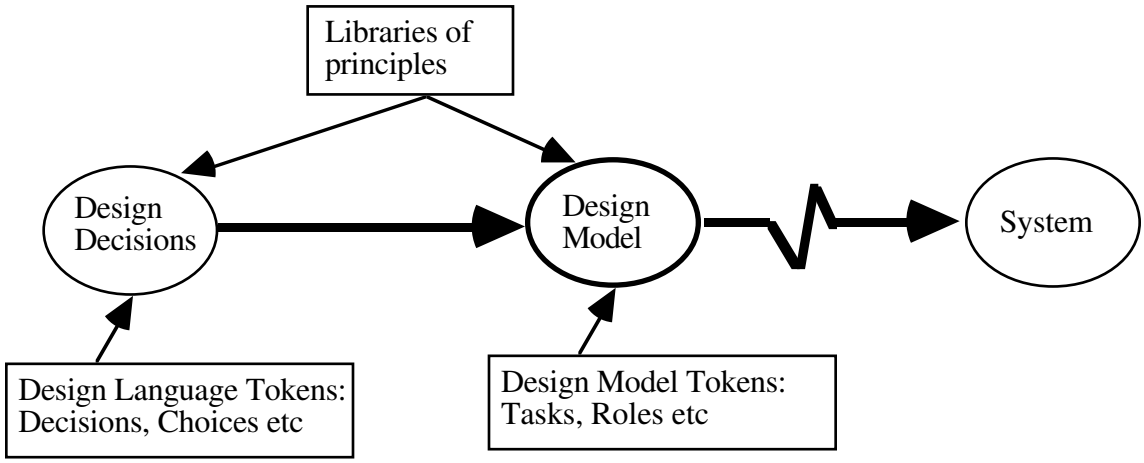
It is this emphasis on knowledge and its representation and manipulation which differentiates KBSs from other software systems. For this reason, following the influential paper by Newell (Newell, 1982), researchers in the knowledge engineering community have proposed representations which describe models of problem solving at the *knowledge level* (Van de Velde, 1994a). In Newell's formulation:

> The system at the knowledge level is the *agent*. The components at the knowledge level are *goals*, *actions*, and *bodies*.…The medium at the knowledge level is *knowledge*…Thus, the agent processes its knowledge to determine the actions to take. Finally, the behavior law is the *principle of rationality*: Actions are selected to attain the agent's goals. (99).

The knowledge level is opposed to the symbol (or implementational) level. If design is to take place at the knowledge level, this means that there must be design problems which are separable from implementation issues. For example task decomposition, the aggregation of design elements into meaningful wholes and the re-structuring of design models are all design decisions which need not have any implementational aspects. However, Newell himself may not have viewed design models as occupying the knowledge level since, as Van de Velde (1994a: 38-4) points out, the models actually used in knowledge engineering research differ from Newell's in that they impose structure on the knowledge. Despite Newell's views, it seems to be necessary to imposes some structure—to provide "a use-specific window on the knowledge level" (ibid.)—if we are to model the knowledge used in problem solving. This structured modelling remains at the knowledge level in that it is not concerned with symbol level issues.

1.3. A knowledge level model of design

We are engaged, in this report, in investigating the structure of a designing agent at the knowledge level. Since we are concerned with the process of producing a design—i.e., with a model of designing—we must include not only the production of the design model by the manipulation of design tokens in relation to design principles but also the decision-making behind this manipulation—the weighing of possibilities, the selection of choices and so on. In Figure 1 we present a knowledge level model of the process undertaken by a rational design agent.



knowledge level **symbol-level**

Figure 1: Knowledge Level Design

Design principles (ordered in hierarchical libraries) are used both to derive the competing choices and to decide on a winner. Further design principles govern the manipulation of the lower-level design components (such as tasks and roles) in cases where choices about them cannot be made at the higher level. Design principles thus play a very important role which is highlighted by their frequent inclusion as justifications in the higher level design decisions (see below).

This approach (with design at a high level influencing the deployment of lower level components) underlies the work of Ramesh and Dhar (1992; 1994) on supporting the design of software systems. In their prototype system (REMAP) high level design decisions affect the lower level design model: “Remap’s reason-maintenance module maintains the dependencies among process knowledge and design artifacts, so that changes in the assumptions, decisions, or requirements underlying a design rationale will trigger the synthesis of a changed design that satisfies the changed constraints” (1994: 57). In the current version of OCML-DDL there is no support for the automatic generation of changes to the design model as the result of design decisions. This is largely because we don’t want to restrict the way designers can use OCML-DDL. It should be up to designers whether they want to work at the decision or the design model level. Nonetheless, some mechanism, similar to that in REMAP, may prove to be essential if only to keep track of changes whichever level they are made at.

2. METHODOLOGICAL BACKGROUND

2.1. Tasks and problem solving methods

Despite the fact that there is some sort of 'rough consensus' in the knowledge engineering community about the need for knowledge level modelling and the role it

plays in knowledge acquisition and KBS design, there isn't yet a reference model or language for representing and discussing knowledge level models which is shared by the whole community. For this reason in this section we briefly discuss the 'methodological background' underlying our discussion.

A knowledge level model can be seen as comprising three loci of interest. These consist of the *task*, *method*, and *domain* perspectives. To our knowledge this view was clearly stated for the first time in (Steels, 1990) and since then - to a smaller or larger extent - all of the most significant contributions to knowledge-level modeling have subscribed to it (Chandrasekaran et al., 1992; Wielinga et al., 1992a; 1992b; Puerta et al., 1992; Runkel et al., 1994).

A task ontology specifies what needs to be achieved. For instance in the VT problem (Yost, 1992) the task is to assign values to the design parameter in a way which satisfies all given constraints. The task can therefore be seen as an instance of a *parametric design problem* (Schreiber, 1994).

A method ontology specifies the knowledge used and required by a problem solving method, which provides a way of tackling a task. For instance, the document describing the VT problem (Yost, 1992) illustrates one particular approach to solving VT, which is called Propose&Revise (Marcus et al., 1988).

Finally, a domain ontology describes information about a particular domain, in a task and method independent way. For instance the VT domain ontology (Gruber et al., 1994) contains information about parts of the elevator and associated constraints, as well as formulas for computing the value of parameters.

A task can be characterised in terms of input and output *roles*, the eventual *constraints* on the knowledge structures which can be referenced by these roles, its *goal* (optional), and the possible *domain requirements* imposed by the task. As an example we show the definition of the class of parametric design tasks taken from a knowledge base (KB) we are currently developing (Motta, in progress), which uses Ontolingua (Gruber, 1993) to formalise in a implementation-independent way our solution to the VT design problem (Motta et al., 1994).

```
(define-class parametric-design-task (?task)
   "A parametric design task is defined as the task of assigning
    values to parameters so that each parameter has a value and each
    constraint is satisfied.
    Parameters can be organised around components.
    The specification of a parametric design task requires that the
    following domain relations are specified in the theory:
    HAS-VALUE (?parameter ?value) and
    HAS-PARAMETER (?component ?parameter)."

   :class-slots ((subclass-of ?d configuration-design-task))
   :instance-slots ((has-input-role constraints
                     components parameters)
                    (has-output-role design-model)
                    (has-goal
                        (and
                           (forall (?p)
                              (=>
                                 (parameter ?p ?task)
                                 (exists ?x
                                   (has-value ?p ?x)))
                           (forall (?c)
                              (=>
                               (constraint ?c ?task)
                               (truth ?c)))))
                    (requires-domain-relation
                     has-parameter has-value)))
```

Figure 2: A sample task specification: Parametric Design

The expression in Figure 2 describes the task of parametric design as taking a set of constraints, components, and parameters as input and producing a design model. The goal is to ensure that each parameter has a value and that each constraint is satisfied. In addition we also state that when instantiating this task in a particular domain, we do not only need to provide domain references for the roles of the task, but we also need to give meaning to the relations 'has-value' and 'has-parameter', which have been used in the task specification (cf. Van de Velde's canonical modeling operators, 1994b: 729). Figure 3 shows an example of mapping the input role 'parameters' of an instance of a parametric design task to a particular domain KB, in this case the Ontolingua specification of the VT domain (Gruber et al., 1994).

```
(define-instance vt-design-problem (application-problem
                                    parametric-design-task)
   :slots ((domain-theory vt-domain)))

(define-axiom reference-for-parameters-role
    "This axiom specifies that in the vt design problem the role
     `parameters'in the parametric design task is mapped to the set
     of all parameter slots in the vt domain."
   := (= (domain-reference parameters vt-design-problem)
       (setofall ?p
                (exists ?c
                      (and
                       (DIRECT-SUBCLASS-OF ?c vt-component)
                       (HAS-PARAMETER-SLOT ?c ?p))))))
```

Figure 3: Mapping a role to domain-level knowledge structures

Problem solving methods are also specified in terms of input and output roles and can be either *primitive* or *structured*. When applied to a task, structured methods decompose it into a number of new sub-tasks, i.e. provide method-mediated task-sub-task decomposition (Chandrasekaran et al., 1992). Primitive methods - also called *mechanisms* (Gennari et al, 1994) - are 'low-level' methods which can be directly applied to solve a task.

```
(define-class Propose&Revise (?method)

   "The Propose & Revise Method takes a set of parameters, a
   set of model extension procedures, a set of constraints
   on the values of the parameters, and a set of procedural fixes
   associated with constraints and returns a target model
   defined as a consistent assignment of values to parameters"
   :class-slots ((subclass-of Composite-PSM))
   :instance-slots ((introduces-subtask
                     Propose-task Verify-task Revise-task)
                    (has-input-role Model-Extension-Procedures
                                    Parameters Constraints Fixes)
                    (has-output-role Target-Model)))
```

Figure 4: A coarse-grained specification of the Propose&Revise method

Figure 4 shows a coarse-grained, functional description of the Propose&Revise problem solving method, which can be used to solve parametric design problems. The method takes as input parameters, constraints, *propose-steps* (also called *procedures*), and fixes, and produces a target model. This is an example of a structured problem solving method, introducing three sub-tasks. Of course, a complete description of a specific *problem solving model*, for instance based on some particular instantiation of the Propose&Revise class of methods, requires i) that problem solving methods are selected to cover each task, and ii) the specification of inter-task control relations.

Finally, we distinguish between two types of roles: *static* and *dynamic*. Static roles denote structures which do not get modified during problem solving, while dynamic roles denote structures which can be modified during problem solving. Dynamic roles further divide into *intermediate* and *solution* roles. Solution roles point to the *case models* (Steels, 1990) which provide the solution to the application problem in hand. Intermediate roles denote instead structures which can change during problem solving but only play a 'support role' during the construction of the solution. In addition roles can be categorised in terms of the source of the information which is referenced by them and we distinguish between four types of sources: *user*, if the information denoted by a role is acquired from the user during problem solving; *problem data*, if it is part of the problem specification; *case data*, if it is information defining the particular test case in hand; and *inferred*, if it is information generated during problem solving. The categorisation described here is based on the approach used in CoE, which distinguishes between static and dynamic roles (which are called 'models' in CoE), and

on the typology of case models in CommonKADS, which distinguishes between initial case data, user input, intermediate case data, and case solutions (Wielinga et al., 1992b).

2.2. The design of KBS—A survey of Methodologies

2.2.1. How does designing KBSs differ from designing other software?

The principal difference between KBS design and other forms of software design is that here the designer is dealing with knowledge. In KADS and VITAL this means that s/he may have to produce a model which may incorporate all or some of the structure of the conceptual model (or model of expertise) produced during the analysis phase. In other forms of software design, even if a conceptual model is produced, it is not regarded as important that its structure is preserved during the design phase. This notion of structure preserving design is a central plank in the KADS-I approach (Schreiber, 1993). It is also important in the KADS-II research. For instance a design model is described as "the computational realization of the expertise and communication model within the boundary conditions specified by organization, task and agent model" (Van de Velde, Duursma and Schreiber, 1993: 11).

2.2.2. A framework for discussing methods for KBS design

Each of the methodologies discussed below is provided with a table which summarises: the theory behind the method, what representations it uses, how it deals with the history of the design process and what tools are included.

2.2.3. KADS I

The design of knowledge based systems using the KADS methodology has been outlined in Schreiber, Bredeweg, Davoodi, and Wielinga (1987) and summarised in Schreiber (1993). A more recent handbook for knowledge engineers also details the KADS-I design approach—Tansley and Hayball (1993). We will discuss the relevant details of each of these below. It is important to note that the model of design presented in all three places is essentially the same. Nonetheless there are differences—if only of emphasis. The 1993 paper is a more developed form of the model first presented in 1987 while the Tansley and Hayball book is aimed at producing a practical and workable introduction to KADS-I for knowledge engineers and is therefore clearer, more systematic but less theoretical. The work described in all three versions is also recognisably the basis for the KADS-II/CommonKADS model discussed below.

KADS-I—The Early Version.

In the earlier expositions of the design methodology, design was viewed as producing a design model which stands at the 'same level of abstraction' as the conceptual model/model of expertise but which is a model 'of the artifact rather than the real world' (Schreiber et al., 1987: 3). Three main steps in the design process were identified: Functional description or decomposition into functions, selection of methods or transformation of functions into design elements, physical description or composition of these into physical modules.

Functional description involves the decomposition of the conceptual model into blocks so as to specify *what* the system will do. Unlike the conceptual model, the functional decomposition must include details of I/O functions and data storage. The decomposition is done in the light of an AI paradigm (such as the production system paradigm) which provides the framework for the decomposition (ibid. 6, 9-13, 40-1).

Selection of methods. This is a specification of *how t*he functions are to be realised. Methods include "natural language interpretation, search algorithms, classification using isa relations, truth maintenance". Design elements are the components of the artefact (or system) which will realise the method(s) chosen. These are illustrated as follows: "a search method like the A* (A-star) algorithm requires a procedure implementing the algorithm, state operators, and heuristic estimates" (ibid. 13-15). The selection of methods will be determined by the paradigm chosen, the type of functional block and the methods associated with knowledge sources in the conceptual model (ibid. 7, 43-6).

Physical description. Here the methods are (re)composed into the modules (and the architecture which they comprise) which is used to produce the detailed or technical design for the artefact/system. An environment (such as EMYCIN, KES or Prolog) for the implementation of the system is chosen at this stage (ibid. 7, 15-19). The process of composition here is guided by "paradigms that prescribe an aggregation of design elements into physical modules in an efficient and non redundant way" (ibid. 47). Note that these paradigms are not the same as the AI paradigms mentioned above but embody general principles of software design such as the avoidance of knowledge redundancy.

KADS-I—The 1993 Version.

In the chapter on design in the KADS book published in 1993—Operationalizing Models of Expertise—the design process remains model centred. While the design model may still be construed as the model of the artefact a greater emphasis is placed on 'structure preserving design' i.e. the principle "that the information content and structure present in the knowledge-level model [model of expertise] is preserved in the

final artefact" (126). The steps in the design and implementation process are shown in Figure 5 which is reproduced from Figure 6.3 in Schreiber (1993). It should be noted that Scriber makes it clear (124) that the figure shows dependencies rather than the strictly prescribed ordering of design steps. Nonetheless it could be used in this fashion.

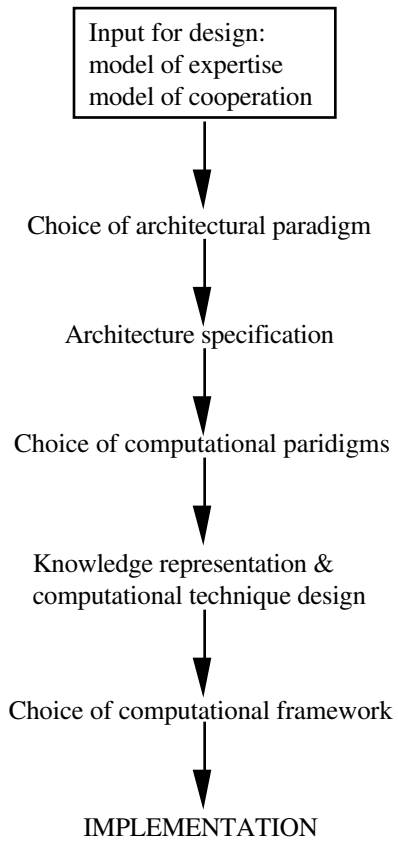


Figure 5: The stages in the design process (after Schreiber Figure 6.3)

Choice of architectural paradigm. At this stage the designer chooses the type of architecture (viewed as a description of the artefact's components and their inter-relationships) for the eventual artefact/system. Schreiber mentions the functional approach of Yourdon, the object-oriented approach of Coad and Yourdon and black-board systems, as examples. The selection of an architectural paradigm determines as well as facilitates later design decisions. Schreiber points to the decomposition principle as the main way of distinguishing these: i.e. in the functional approach the system is decomposed into functions and their sub-functions, in the object-oriented approach data objects are the focus while in the blackboard approach control is the basic issue (121).

Architecture specification. Depending on the choice of paradigm the designer will now specify the architecture. This is aided by the provision of skeletal architectures or "prototypical decompositions which have to be instantiated for a particular application"

(122). In fact, following the principle of 'structure preserving design', the skeletal architectures given include components which reflect in the design model the relevant aspects of the model of expertise. This, therefore, is put forward as a new architectural paradigm which does not emphasise data, function or control but which includes all three.

In brief, the new architectural paradigm reflects the 4-level model of expertise in which domain, inference, task and strategic knowledge are modelled. The skeletal architecture (intended to preserve this structure) includes:

- a domain knowledge base and its access functions (which require indexing mechanisms to map from meta-classes to actual domain items);
- various inference functions (the design equivalent of knowledge sources) which are realised with inference methods (see below);
- a task interpreter which will act as the central control of the artefact/system;
- meta-level reasoning for strategic task, inference and domain level knowledge;
- various functions for inter-acting with users.

Choice of computational paradigms. Here the designer chooses (for the various components in the architecture) between well-defined paradigms such as production systems, automated deduction, state-space search, parsing, constraint satisfaction and structure matching (123). These in turn determine choices about knowledge representation and computational techniques.

Knowledge representation and computational technique design. The knowledge representation will either be given by the computational paradigm or derived from the model of expertise. Computational techniques (which realise inference methods) consist of algorithms and input-output data structures as well as the representations of domain knowledge (134).

Choice of computational framework. Finally the designer has to choose an environment to be used to implement the system.

KADS-I—The Version in Tansley and Hayball.

The KADS-I methodology has been tidied up for knowledge base designers and developers in the handbook produced by Tansley and Hayball. The Tansley and Hayball book usefully synthesises the two papers summarised above.

According to Tansley and Hayball, there are two main stages in design: Global and KBS Design. In the former a Global System Architecture is produced. This is "a model that describes the major sub-systems in terms of which functions they perform and how

they interface to the other sub-systems" (128). For each of the KBS sub-systems a process of KBS Design is performed. This process can be divided into four stages each of which produces a result as indicated below in Table 1. This table (after the tables given on page 178 of Tansley and Hayball) shows the generic (software engineering oriented) stages, their result and the actual KADS activities:

KADS Design Stage	**Result of Stage**	**Actual KADS Design Activity**
GLOBAL Design	Global System Architecture	
Define KBS Design Framework	KBS Design Framework	
Functional design	Functional Design Model	Decompose KBS Functions
Behavioural design	Behavioural Design Model	Assign Design Methods
Physical design	Physical Design Model	Assemble Design Elements

Table 1: KADS Design Stages and their results (after Tansley and Hayball)

KBS Design Framework. This is "a general problem-solving technique, such as constraint-based logic or blackboard reasoning. Alternatively, it could be related to the chosen or dictated implementation environment, e.g. frame- or rule-based" (145-7). The framework thus takes account of constraints on the design of the system and acts as the context for other stages in the design process.

Functional Design Model. This model is "a hierarchical structure of the functional components of the system. These components, known as *function blocks*, are derived form a transformation of the analysis material. For components that store information, the equivalent to function blocks is the *domain model*, which details the data and knowledge to be stored" (128). In addition to the problem solving and storage functions there are Strategic Control functions (which specify control within the KBS function) and Support functions (which include other functions such as "explanation and housekeeping") (150-1). The functional design model transforms the strategy layer of the expertise model into strategic control functions, the task layer and inference layer into problem-solving functions and the domain layer into a domain model.

Behavioural Design Model. This is a "description of the computational and representational approaches which have been chosen to realize the required problem-solving functions, i.e. what actual methods/structures will be used to realize these functions" (162). It thus depends on the functional design model (and indirectly on the model of expertise). Within the behavioural model *design methods* are chosen to realise function blocks while *design elements* refer to the algorithms and data structures which comprise these. Knowledge representations (for domain models) are also chosen at this

stage. Tansley and Hayball provide a library of problem solving methods and representations.

Physical Design Model. This model is the result of re-arranging the design elements and knowledge representations (mentioned above) into modules of code. At the same time an architecture (showing the relations between these modules) is specified (171-2).

Theory	Design Theory	SE, Ad hoc
	Process model	Yes
	Principles of design	?
Representations	Formal design language	DDL
	Informal/Graphical design language	No
History	Design rationale	No
	Design clichés	Design elements, PSMs
	Design cases	Experiments, Examples
Design tools		No ?

Table 2: Summary of KADS-1 design methodology characteristics

2.2.4. CommonKADS/KADS II

In the continuation of the KADS-I project, KADS-II, work is being pursued to update the KADS-I models in the light of other research in knowledge engineering. The principle result of this has been the CommonKADS framework for knowledge modelling (Wielinga, Van de Velde, Schreiber & Akkermans, 1992b) which incorporates some of Steels' ideas as presented in the Components of Expertise approach (see below). In this framework there are three main categories of knowledge (domain, task and inference) and problem solving knowledge. This latter is "not another knowledge category in addition to domain, task and inference knowledge. Problem solving knowledge is about a 'meta application', namely to come up with a good model in terms of application related knowledge" (Wielinga et al, *31*-14). As the authors go on to suggest, the *task* of problem solving knowledge is modelling, the [*inference*] method is method selection and application and the *domain* is problem solving methods and strategic knowledge (ibid.). A problem solving method "specifies the relation between a task definition and a task body" (ibid., 31-16); i.e., a problem solving method supplies a means of filling out (by further decomposition into sub-tasks or by some primitive or basic task) or 'instantiating' the task body of a task which meets certain criteria for the application of the problem solving method. As with Steels, the dynamically constructed model (application of PSM to task which operates on domain via inferences) constitutes a *case* model. Wielinga et al. provide a formal definition of a PSM (31-16):

"A problem solving method is a prescription of the way in which a certain class of task definitions can be satisfied. The method may be complete with respect to a set of basic

capabilities, i.e. prescribe all necessary actions in terms of the basic capabilities of the agent, or may be incomplete in the sense that the activities generated by the method may lead to new problems, which cannot be solved by basic capabilities and require the application of new problem solving methods."

The CommonKADS *design model* is presented in Van de Velde, Duursma and Schreiber (1993). It is important to note that the work presented there is very much in progress and that significant elements have not yet been developed (e.g., the design library and the details of the design process). Other aspects such as the design language are reported in Terpstra and Schrooten (1994) which also to some extent covers the design process. Van de Velde et al adopt an approach to KBS design which develops the ideas formulated in the KADS project (Schreiber, Bredeweg, Davoodi and Wielinga, 1987), remaining true to the KADS modelling approach while investigating links with more conventional software engineering.

For Van de Velde et al (11) "the design model describes the computational realization of the expertise and communication model within the boundary conditions specified by the organization, task and agent model. More precisely it is a model of a problem solving behavior in computational and representational terms." They view design as (11-12) "the process of creating a software design that fits the constraints and requirements that are implicitly or explicitly contained in the relevant pieces of the CommonKADS product model, i.e., the organizational models, the task models, the expertise models, the agent models and the communication models." Among the other considerations which they advance for the importance of the design model are included (14): a) the good practice of separating analysis and implementation; b) the platform independence of the design model; c) re-usability.

The design model has three constituents: application design, architecture design and platform design. Each constituent is composed of two parts. In the case of application and architecture a specification is 'realized by' a 'detailed' design. The platform design has a language specification which is 'realized by' an 'implementation platform'. The detailed components of these constituents are said to be 'realized within' the specification components of the succeeding constituent. These features are detailed in a design model template (Section 2.4 of the Van de Velde et al report). In the following we have tried to summarise the more important features of this template.

• Application design "operates at the conceptual level of 'real-world' entities and tasks that have to be realized by the artifact" (15) and involves the decomposition of the artefact (the system being designed) into 'sub-systems' (which realise tasks from the model of expertise) and an application architecture which 'describes the top-level

control between the sub-systems' (18). It also includes reference to a design description language (HADL a high level application design language is detailed in Terpstra and Schrooten, 1994). The detailed application design is related to the architecture design which provides the language for a 'program' (the detailed specification of the sub-system) which will run in the architecture. In the example given in Terpstra and Schrooten, 1994 the detailed application design includes a program which is expressed in the set manipulation terms given by an architecture for handling sets.

• Architecture design "refers to the abstract computational structures within which the application design is realized" (15) and is specified in terms of computational objects (such as sets), computational methods (which operate on the objects) and the relations between them (e.g. the method *remove-element* has the relation *WORKS-ON* to an element and a special set). The detailed architecture design specifies the computational objects and methods in a program description language (derived from or identical with the language specified in platform design).

• Platform design "specifies the target implementation language and the hardware and software platform that will be used" (15) and includes details about the user and user environment (25).

In terms of the VITAL distinction between types of design, the application design constituent (or at least its specification component) of the design model corresponds to Functional design while architecture design, platform design and detailed application design (depending on the level of detail) correspond to Technical design.

While the process aspect of design is still being researched Van de Velde et al. give a useful list of the 'development states' of the model and their time ordering (32-36). In addition they provide a 'preliminary' list of 23 modelling guidelines (38-9)

Theory	Design Theory	SE, Ad hoc
	Process model	Yes
	Principles of design	Guidelines
Representations	Formal design language	HADL
	Informal/Graphical design language	No ?
History	Design rationale	No
	Design clichés	Library of design components
	Design cases	Examples
Design tools		Editor?

Table 3: Summary of KADS-II design methodology characteristics

2.2.5. Components of Expertise (CoE)

in the Components of Expertise method (Steels, 1990, 1992) there are three main components: models, methods and tasks. Steels suggests that "the three perspectives are worked out in a kind of spiral movement each time progressively refining the description from a particular perspective" (1992:2). He goes on to suggest how the knowledge engineer might utilise the CoE method: "…the designer would first think about the task, then move to a model perspective to work out some of the models involved in the task, then think about the methods to achieve the task which typically means that more models or more sub-tasks are introduced, and so on" (1992:8). As his use of the term 'designer' in the above suggests, Steels does not make a sharp distinction between knowledge analysis and design—indeed he could be said to suggest that design is the more general task which includes analysis. This is borne out on pages 14-15 of his memo where he pictures the development of an applications as follows:

analysis —> design —> knowledge acquisition —> application execution.

Figure 6: Application Development in CoE

Both analysis and design are preliminaries to acquisition which may lead to more analysis and design. As he says: "…the first two activities (analysis and design) are very much intertwined and there is a spiral top-down refinement of the different descriptions" (1992: 15). Nonetheless design can be characterised as including (ibid.:15):

(a) the selection of "the form of the different models";
(b) the finding of the "methods that could accomplish a particular task";
(c) the filling in of the roles included in a method.

Selecting the form of a model. For Steels a model can either be a *case* or *domain* model. The former are built up during the design of a particular application, the latter are fixed. In his example of a motor-cycle fault diagnosis program, the domain models include those for faults and mappings between symptoms and faults. The case model here is the particular set of symptoms. The form of a model means the particular type of structure which is utilised: Steels mentions sets, sequences and sets of mappings. The designer must therefore decide upon the particular structure for the relevant models.

Selecting methods. "A method," says Steels, "is an algorithm" (1992:11). It defines a control flow over a set of activities which use models as input and change case models. These input and output models are said to play a particular *role*. Some methods may involve further sub-tasks and hence require the selection of further methods and so on.

Filling in roles. The designer may be involved either in acquiring the details for the domain and case roles or in defining new roles (and selecting their form).

In general the components are selected from an *application kit* (or built from the primitives contained therein). Application kits may be developed for various applications such as diagnosis. Steels does not provide much in the way of guidelines for selecting components though he does say "Selecting a method is a non-trivial task. It depends on the task features, available domain knowledge, as well as decisions about the form of models" (1992: 16).

Theory	Design Theory	SE, Ad hoc
	Process model	Yes, cyclical
	Principles of design	?
Representations	Formal design language	No
	Informal/Graphical design language	No
History	Design rationale	No
	Design clichés	Design components
	Design cases	Examples
Design tools		KREST?

Table 4: Summary of CoE design methodology characteristics

Comments

Steels eschews the clear distinction between analysis and design and suggests that acquisition takes place after design. This is inadequate for the following reasons:

1. Before any meaningful analysis or design can take place the knowledge engineer must do some preliminary knowledge acquisition in order to contextualise the subsequent analyses and designs.

2. In many cases it is useful (if not necessary) for there to be an analysis model (conceptual model, model of expertise) which captures domain knowledge in a manner independent of any computational issues. Design (of KBSs) is concerned with finding ways of making use of domain knowledge to solve problems in a computational framework (even if divorced from specific details of an implementation). As such, there might be several design models (and many more implementations). This is because there are many ways of solving design issues. If, as the CoE approach suggests, the design and conceptual models are not separated, there will be no single, definitive model of the domain (whether this is called the conceptual model or the definitive or ur-design is unimportant).

2.3. The VITAL knowledge engineering methodology

Here we make use of the VITAL knowledge engineering methodology which characterises KBS development as the production of four process products: the *requirements specification*; the *conceptual* model; *functional* and *technical design models*, and the *executable code*. A conceptual model is the output of the *knowledge analysis* activity and provides a knowledge level description of the domain knowledge, and the problem solving methods used to solve the given application tasks. A functional design model is a knowledge level description of the target system. The same modelling language can be used to describe functional design and conceptual models, as these only differ in their modelling focus. The former describe the problem solving strategies used by the target system while the latter describe those used by domain experts. In some cases these can be the same and therefore there is very little difference between the two models. Typically the functional design model will also include some communication tasks such as those needed to acquire data from external sources. In other cases KBS designers might decide to implement problem solving methods which differ from those used by the expert. For instance - as discussed in our solution to the VT problem (Motta et al., 1994) - the problem solving behaviour described in (Yost, 1992) can be characterised as a special case of the Propose&Revise method, which we dubbed 'Extend Model and Revise' (EMR). However, when designing our own version of the VT system, we devised alternative problem solving strategies and we found that some of them were more efficient than the one used by the expert (i.e. more efficient than EMR). The explicit recording of both models of problem solving (the expert's and the system one) allowed us to carry out more in-depth studies of Propose&Revise problem solving (Zdrahal & Motta, 1994) and - in a real application development scenario - could contribute to recording the design rationale thus increasing the maintainability of the application.

During technical design, the designers decide how the knowledge-level structures specified in the design model are going to be encoded in implementation-level structures. This activity corresponds to what is called 'design' in the KADS-1 methodology (Schreiber et al., 1987).

Finally, the executable code is the result of the implementation activity. It comprises all software modules which are part of the target application.

The VITAL methodology is supported by a workbench, which is described in (Domingue et al., 1993).

In the context of this report - unless specified otherwise - we will use the term 'design' to refer to the activity of knowledge level modelling. Within VITAL, this activity subsumes both functional design and conceptual modelling.

3. DESIGN AND ARGUMENTATION

Design is an activity which is by its nature non-monotonic i.e. decisions are made and changed in the light of new considerations. The propose-evaluate-revise schema used in AI approaches to design (Chandrasekaran, 1990) makes explicit this non-monotonic element in the design process.

One way of looking at this non-monotonic characterisation of the design activity is as an example of an argumentation process. The proposal can count as a *claim* which is supported by *grounds* and *warrants*. Evaluation can be thought of (at least partly) as the evaluation of an argument and revision as the linking of new claims to their supporting grounds. In fact while the propose-evaluate-revise schema captures the process at a coarse-grained level it is also active at arbitrarily finer-grained levels. The making of a proposal can be decomposed in sequences of propose-evaluate-revise at lower levels as each part of the design is selected tentatively, replaced and finally settled upon before the final proposal is complete and the design as a whole can be evaluated.

Argumentation (at least with one's self) can thus be seen to form part of the cognitive processes involved in design. Argumentation (in the sense of explicitly represented arguments) has two further advantages:

- As with any such activity, it is useful as a means of book-keeping, in particular for maintaining an account of the decisions made and their rationale (this is particularly true when more than one designer is involved)—i.e. it contributes to the design *history*;
- It can play a role in making the design (and its rationale) more comprehensible, coherent and cogent—i.e. it can form part of the design *representation*.

While we feel that there is much to be gained by the use of argumentation in design (especially in situations where either a single designer is working over a long period or where there are multiple designers) there are opposing views. Buckingham Shum and Hammond (1994) explore the extent to which the use of systems to record the design rationale are either usable or helpful. They conclude that not enough empirical evidence has been collected to come to any conclusions. But, unless we create such a system, no empirical work can be done. The best solution is to base the tool on an empirical study as reported by Ramesh and Dhar (1992; 1994). The problem then becomes one of

deciding whether the results from a particular experimental domain are sufficiently generalisable or whether a similar experiment needs to be done for each domain.

3.1. The Design Rationale

Different theorists have slightly different views of the design rationale. Buckingham Shum and Hammond (1994: 604) suggests that "a design rationale expresses elements of the reasoning which has been invested behind the design of an artifact." Fischer et al. (1991: 395) point out that in their approach "design rationale means statements of reasoning underlying the design process that explain, derive, and justify design decisions.…Design rationale ..... is a synonym for argumentation". For Gruber and Russell (1991: 2) "A design rationale is an explanation of how and why an artifact, or some portion of it, is designed the way it is." We take the design rationale to be a space of actual and alternative choices, the relations of attack and support between these, with supporting chains of justification and possible rebuttals, attached to design elements or the design as a whole—the design rationale is a representation of the totality of the argumentation which went into the making of design decisions. At each step in the design process the design model is augmented (or changed) in two ways: the design elements are transformed and the (relevant node in the) design rationale is updated—though not necessarily automatically as in Ramesh and Dhar's system (1992; 1994).

The literature (Gruber and Russell, 1991; Fischer et al., 1991; Buckingham Shum and Hammond, 1994) abounds in lists of reasons for recording the design rationale. The important points from these can be summarised as follows:

- The design rationale can record design decisions for a single designer over time or for a team of designers.
- The design rationale can be used for inter-designer communication.
- The design rationale can be used to communicate ongoing work to managers, clients and others not in the design team.
- The design rationale acts as a form of mental hygiene, tightening up ideas and showing up misconceptions, and can help in overcoming mental blocks.
- The design rationale can act as a means of teaching the house style of a design organisation to new designers.

This last point is particularly important for Van de Velde's modeling libraries since the model is to serve as "a common ground of discussion" (1994b: 730).

The use of the design rationale is particularly well suited to the design of KBSs since, lacking well-defined theories of human expertise, design decisions can be particularly

idiosyncratic. The design rationale documents what decisions are made about which design component and why. As such, the design rationale captures, not only Van de Velde's "best practices" but also forms the basis for the induction of new design principles.

3.2. Non-argumentative design

In some cases, for example, when there are no competing arguments or when a step-wise design methodology is being rigidly pursued, the design rationale simply records the decisions made without supporting arguments. Design here is just the structuring of high level components in the design process (decisions, faults, fixes, constraints) without the addition of argumentational structures.

3.3. Models of Argumentation

There are several models of argumentation discussed in the literature (from areas such as computational modelling of language as well as design). Models from the design literature are summarised in Jarczyk et al. (1992), Buckingham Shum and Hammond (1994), Ramesh and Dhar (1994). Stutt (1989) provides an overview of computational approaches to argumentation in general, which is briefly summarised below.

Toulmin (Toulmin, 1958, Toulmin, Rieke and Janik, 1979) views an argument as a complex data structure in which a claim is related to the evidence (or data) it is based on by means of a warrant (a piece of knowledge allowing the derivation of the claim from the evidence). This warrant can be of many different types corresponding to the mode of inference used. In turn, the warrant is grounded in a backing drawn from theoretical or common-sense knowledge. This multi-dimensional mechanism ensure that the Toulmin model, unlike the alternatives, provides a means of specifying the different types of inferences (analogical, inductive, deductive etc.) and their backing theories. This model can be extended to include design methodologies as backing.

Since Toulmin's model only deals with single claims (or argument steps) the basic model needs to be augmented to deal with the relationships between the argument claims. Stutt's *extended Toulmin model* (Stutt, 1989) identifies the basic Toulmin structure as level 0 in a three level model which also includes:

- Argument-1s. Each argument for or against a proposition in a debate can have a tree-like structure in which propositions at the root node are supported by arguments and sub-arguments represented by nodes at the various lower levels.
- Argument-2s. This top level represents the debate as a whole and involves the representation and manipulation of a complex data structure which records the arguments for and against particular propositions by particular participants.

In the *IBIS method* (Issue-Based Information System)[1] there are three possible node types—issues, positions and arguments. Issues or questions are answered by positions each of which has a set of pro and con arguments. Various relationships link the issues. Fischer et al. (1991: 397) mention "more general than", "similar to", "replaces", "temporal successor to", "logical successor of". Jarczyk et al. (1992: 579) mention "supports, objects-to, replaces, responds-to, generalizes, specializes, questions, and suggested-by."

While Rittel's method has been used in a computer system directly most of the work on implementing IBIS systems has used variations on the model. In particular there is the *PHI* (Procedural Hierarchy of Issues) approach due to McCall (McCall, 1991; Fischer et al., 1991). In essence this differs in that there is only a single *serve* (subissue-of) relationship possible between issues. Issues are therefore arranged in hierarchies of issues with their answers (positions) and arguments. The PHI approach forms the basis for the various tools emanating from the design group at the University of Colorado (Fischer et al., 1991 [kitchen design]; Fischer et al., 1989 [kitchen design]; Fischer et al., 1992 [software design]; Lemke, 1990 [computer interface design]; Eisenberg and Fischer, 1994 [chart design]). A similar extension has been made by Conklin and Begeman (1988) in their gIBIS tool.

3.4. Approaches to design and argument

It is possible to differentiate between the various approaches depending on how close the coupling between design and argumentation is. In the most loosely coupled systems such as gIBIS (Conklin and Begeman, 1988) a generic hypertext based on the IBIS approach is utilised as a means of articulating the design rationale.

In systems such as JANUS (McCall, Fischer and Morch, 1990) which integrates the CRACK critiquing design system with an argument environment and JANUS-ARGUMENTATION (Fischer et al., 1991) an attempt is made to produce an architecture which embodies Schön's ideas on design as "a continual alternation between two radically different and mutually exclusive types of design processes: knowing in action and reflection in action" (ibid. 408). The former being the non-reflective interaction with the designed artefact while the latter is needed when some 'breakdown' between what happens as the result of a design step and the designer's expectations. In addition to a 'construction kit' of design components and a hypertext for the design rationale, JANUS-ARGUMENTATION includes a library of design examples, a means for designers to specify (partially) their goals and a simulation

[1] While Rittel has written generally on design as "an argumentative process" (Rittel (1984: 320)), the original report on IBIS is not readily available. We therefore rely on secondary sources for our summary of the method.

module. A set of software tools mediates between the various components in this architecture.

In Remap (Ramesh and Dhar, 1992; 1994) a close connection is maintained between the design rationale and the design objects— "The decisions imply, generate, or lead to design constraints, which cause design objects to be modified, created, or removed from the design solution" (ibid. 55). Design is thus primarily a process of argumentation with indirect manipulation of design objects.

In KONEX+ (Rathke and Negele, 1991), a design environment for CNC-machines 'design experts' are utilised as cooperative agents which can provide design advice and critique user designs.

In Gruber and Russell's approach (1991), design rationales are not stored but are automatically generated by the systems they discuss on demand. IDE 2.0 generates rationales from design principles while DME uses simulations of device models. From our point of view, since we seek to support human design activities, the automatic production of what amounts to a high level *explanation* of the system's reasoning is beside the point. We are interested in how human designers design and in the rationales they actually proffer.

4. LANGUAGES FOR KBS DESIGN

Various design languages have been proposed for the various design methodologies. Design languages are used as a means of i) clarifying and debugging a design and ii) communicating this design to others. In addition the use of a computationally tractable representation provides the possibility of the automatic implementation of the design model.

4.1. Languages for KBS design

There are two main types of design languages mentioned in the literature: formal languages - for instance $K_{BS}SF$ (Jonker and Spee, 1992; Jonker and Motta, 1993) - which have some basis in formal specification techniques in software engineering, and semi-formal approaches - for instance OCML (Motta et al., 1994) - which have a more practical goal and are based on various representational formalisms drawn from artificial intelligence or on graphic notations.

• $K_{BS}SF$, the formal design language introduced in the VITAL project specifies data by means of Abstract Data Types and knowledge by means of an order sorted predicate logic (Jonker and Spee, 1992; Jonker and Motta, 1993).

• VITAL-OCML, an informal implementation oriented approach (with a graphical equivalent) used in an earlier phase of the VITAL Project is based on the VITAL conceptual modelling language (Motta, O'Hara, Shadbolt, Stutt, and Zdrahal, 1994, Jonker and Motta, 1993). It includes the definition of concepts, relations, tasks, roles and rules.

• DDL is the design language used by Tansley and Hayball (1993) and is "based on a series of frames". This is not to be confused with the Domain Description Language discussed in Schreiber (1993). Tansley and Hayball refer to a Domain Modelling Language for the same purpose (1993: 72).

• HADL is the design language used in CommonKADS (Terpstra and Schrooten, 1994). It is an extension of the KADS-I CML (Domain Description Language + Knowledge Sources, Tasks) and $(ML)^2$ (a more formal logic based language).

• Krest (Steels, 1992) is basically an environment for developing lisp-based prototypes of KBS applications. It provides a graphical language for specifying data and control flow in the task layer and enables the user to link the task layer to domain structures represented in lisp. It also supports the creation and reuse of libraries of components.

• Ontolingua. Strictly speaking Ontolingua (Gruber, 1993) is not a language, but a tool for building ontologies. When building an ontology the Ontolingua user first chooses a target language—by default this is KIF (Genesereth and Fikes, 1992)—and then develops his/her model by making use of a number of constructs to define relations, functions, logical axioms, etc. The main strength of Ontolingua—in particular when using the KIF implementation—is that it provides i) a very expressive language for KBS modelling, and ii) a mechanism for factorising ontologies into theories and for building new theories on top of existing ones. Moreover, translators from Ontolingua to a number of knowledge representation languages have been implemented and Ontolingua is rapidly emerging as the de facto standard for defining ontologies.

• KARL. The design philosophy in KARL (Angele et al., 1994) has many points with that of OCML. In particular both KARL and OCML are operational KBS modelling languages and are therefore suitable for rapid prototyping of knowledge level models. Moreover, both KARL and OCML are part of comprehensive knowledge engineering frameworks providing methodological support for KBS development. Finally, both KARL and OCML support formal as well as informal modelling. The main difference between KARL and OCML has to do with the fact that the main goal of the KARL language is to provide a formal specification language for KADS models, while OCML pursues different objectives. Firstly, it is based on the 'tasks vs. methods vs domain' distinction, rather than on the KADS four-layer framework. Moreover it aims to

provide an operational, upward-compatible subset of Ontolingua for which 'effective' compilers to commercial delivery platforms, such as CLIPS and KnowledgeWorks, can be produced without too much effort.

While much is gained by the use of these languages, especially in terms of tractability and clarity, they nevertheless have various shortcomings:

- Both formal and semi-formal design languages are less useful for working out the design (especially in the early stages of a project) than for representing the final design. Graphical design environments—tools which embody one or more representational schema into which completed designs can be translated—are more useful here.
- None of these languages includes any means for recording the design decisions (and their rationale). This is especially important where a large team is involved in the design of a KBS. But even in the case of a single designer, since design is a process which may stretch over a long period with the resulting de-familiarisation, some form of recording of the rationale is needed. This is true both of the formal and informal languages as well as graphical environments based on them. We suggest that the formal and informal languages for design can be extended to include structures for argumentation about design decisions.
- One problem with attachment of design rationales suggests a further problem with design languages. When we come to attach the rationale we will be faced with a choice of design language component to attach to. In the case of a decision about one type of task rather than another or one type of role rather than another we can simply attach to the task or role. However, we may often have in mind, as designers, various completely different ways of completing a design; we may, for instance, have decided on this set of roles and tasks rather than that. This suggests that design languages should include some means of combining design elements to form chunks which have meaning for designers. These might take the form of *clichés* (Rich and Waters, 1990) or blocks which are frequently re-used in different designs (or perhaps only by this particular designer). The attachment can then be to the chunk or cliché. Even so, it is likely that there will be cases where it is inappropriate to attach the rationale to a particular chunk. In this case a top level attachment can be made to the KBS design as a whole.

4.2. A decision based design language for KBS design

To overcome these problems (and in keeping with Van de Velde's proposal for modeling libraries), we have developed an extension to the OCML language used for knowledge-level modelling in VITAL, called OCML-DDL, which includes a design

rationale, a design representation and a specification of design elements. Lee and Lai's (1991) generic design rationale language—Design Representation Language—is a comparable approach.

- The *design rationale* is based on a combination of the IBIS approach and Stutt's extensions to Toulmin's model—it records the reasons for the choice of design elements etc.;
- The *design representation* is a sequence of sentences (or their graphical equivalent) which represent the well-formed (syntactically valid) combinations of design elements;
- The *design elements* are the basic constituents of the design—the pieces which the designer manipulates to produce a KBS design. They are integrated into the design rationale by means of :object and :outcome slots in the design decisions.

OCML-DDL is a task oriented modelling language which seeks to support the *process* of designing KBSs as well as the recording of design rationales. Whereas an alternative language like $K_{BS}SF$ is intended to capture the design components (tasks and roles etc.) in a formal manner, OCML-DDL is a semi-formal means of depicting and manipulating the design model at the knowledge level. Where Ontolingua provides a high degree of portability, OCML-DDL has several advantages:

- OCML provides not just a language but a framework and a methodology for KBS development. Moreover, it is part of a workbench providing full life-cycle support for KBS development.

- In contrast with OCML, Ontolingua is not suitable for rapid prototyping. This makes it more difficult to verify and validate an Ontolingua KB.

- Although Ontolingua provides translators to a number of different knowledge representation languages, none of these is (obviously) as powerful as Ontolingua. Therefore translation from Ontolingua to a target language is necessarily partial. This can cause problems to the user especially because there is only limited support and documentation telling him/her which constructs are operational and which are not.

The model for design argumentation (as presented in Figure 7) combines aspects of the IBIS and Toulmin models of argumentation with aspects of Gruber and Russell's theory (cf. Gruber and Russell, 1991: 13) and the work by Ramesh and Dhar (1992; 1994). Each design decision (an *Issue* in IBIS terms) is based on a decision between two or more competing arguments (a single argument needs no meta-decision). Each argument is composed of a proposal (as in IBIS) grounds, their warrants and backings (as in Toulmin). In general, proposals are about the choice points in the design method

and are brought about by conflicting consequences of applying design principles (where there are applicable principles available). The principal features of this model are:

- The introduction of an explicit decision about competing arguments as a means of making a design decision.
- The arguments are contextualised in that they are about particular *issues* or design decisions: i.e. areas of the design which need to have decisions made about them.
- The supports and attacks links between competing arguments can be recorded as a means of documenting their point and aiding their assessment.
- The semantics of the design model is given by the arguments recorded in the design decisions.
- Unlike the model given by Gruber and Russell, there is no automatic means of generating the arguments. The Argument Decision is a means for recording a human decision about the force of the competing arguments and, hence, about the best design decision.
- Unlike the model given by Ramesh and Dhar, changes in the argumentation are not automatically reflected in changes in the target model.

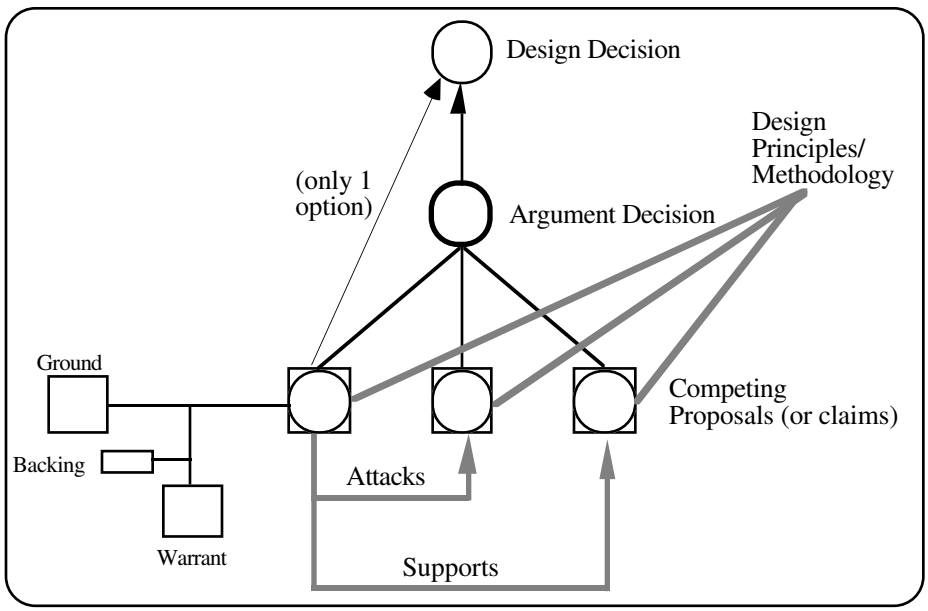


Figure 7: A model of design argumentation

In the VITAL approach to the design of KBSs, the designer follows a series of design steps some of which involve choices between competing possibilities. Choices are largely made on the basis of design principles and must take into account any operational design constraints.

As shown in Figure 8, proposals, choices, fixes and constraints are derived from Design Principles (constraints also derive from Requirements Specifications, Expert Problem Solving Models and Domain Ontologies). Design Steps are also derived from Design Principles (of the form "If step n has been completed go to step n+1"). We make distinctions between them in Figure 8 for explanatory reasons.

At its simplest, design can be seen as the application of principles to a partial model to produce a new model (cf. Gruber and Russell, 1991; Van de Velde, 1994b). However, it better to see the process as the application of principles during design steps to produce (and choose between) proposals and identify and fix design faults.

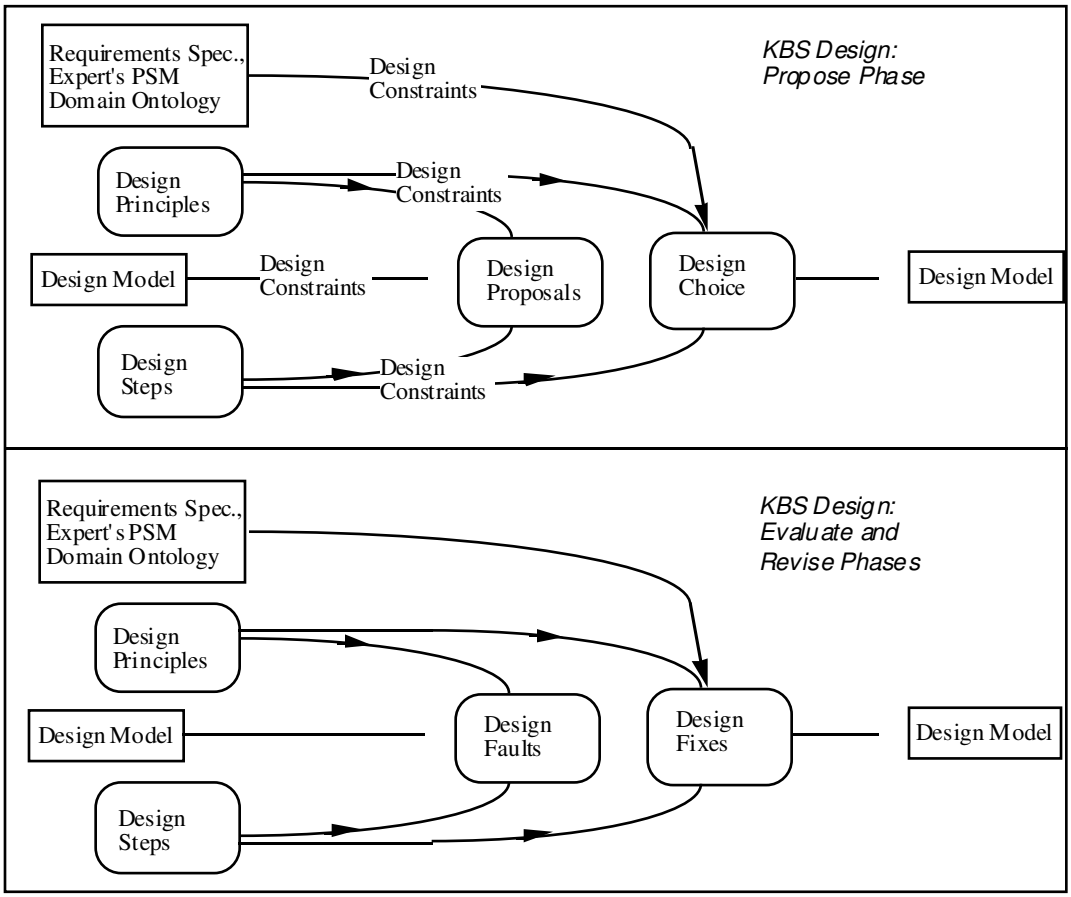


Figure 8: Design as argumentation

4.3. Design Principles—the form of the library

Van de Velde (1994b: 728) gives the following definition for his modeling operations: "A modeling operation is an operation (relation) over models that transforms a generic model into another one". He goes on to suggest that "Minimally they are links between generic models that document that one model can be viewed as the result of some operation applied to another model". An operation is also seen a "modelling step". Van de Velde has various categories of operator: canonical, composition, transformation and modification.

Our design principles fulfil most of the roles of Van de Velde's operators while making these roles explicit. Design principles are intended primarily as guidelines to the designer. However they are also indexed so that relevant principles can be made available at any point in the design as a means of aiding designers in their decision-making or to tell them what to do next (i.e. what (possibly transformative) design step to take). Thus, they are not necessarily operational but do often act as (meta-)rules for making a choice between competing design decisions or for transforming one design model into another. When they are actually deployed they are recorded in the design rationale and thus act as a means of documenting a particular design choice and showing how and why a particular design component may be (re)used.

Our design principles are multiply indexed in a library along various dimensions (which subsume Van de Velde's typology).

- *Principled Activity*. As suggested above, design principles can act as: design steps (producers of new model expansions); facilitators for choosing between these; constraints (on design choices, on relations between design components); evaluators (using constraints to identify design faults); fixes (of faults).
- *Results from.* Principles are contextualised so that relevant guidelines are made available as a result of the carrying out of a design step (see below for an outline of these steps).
- *Attached to*. In addition, sets of guidelines are 'attached' to (partial) problem solving methods (PSMs) so that when a PSM is chosen the appropriate guidelines for future choices are readily available. As with the previous dimension, this does not mean that the designer is restricted to the use of this particular set. All principles are always available.
- *Level*. Principles act to guide design decisions at various levels ranging from the very general to the specific: when choosing between design methodologies, problem solving methods, tasks, and so on.
- *Source*. Principles can be indexed to indicate their source: software engineering practice, individual knowledge engineers, KE methodologies, experts, clients and so on

Principled Activity	Results from	Attached to	Level	Source	Example
Any	Any	Any	General Model	Software Eng.	"Always make least commitment"
Any	Any	Any	General Model	Software Eng.	"Satisfy the requirements spec."
Any	Any	Any	General Model	Software Eng.	"Always make least changes"
Any	Any	Any	General Argument	Informal Logic	"If there is no means of deciding between two proposals and it is possible to combine them, do so"
Evaluation	Propose or Fix	Any	General Model	Design Theory	"If there are unfulfilled constraints from the requirements and domain data then these must be obeyed even if this means that extensive revision of the design model is needed"
Design Step Proposal	Any	Any	General Model	VITAL	"In general expansion is depth-first left-right"
Design Step Proposal	Any	Any	Method Selection	VITAL	"Always use the more comprehensive method if no reason to use less comprehensive"
Design Step Proposal	Propose step	Any	Next step	Design as Propose, Evaluate, Revise	"When something is introduced into the design model it needs to be evaluated."
Design Fix	Evaluation	Any	Next Step	Software Eng.	"In order to fix the redundant application of a task impose conditions which need to be satisfied before the task is invoked"
Design Step Proposal	Propose step	Propose1 PSM	Task expansion	VITAL	"If the there are no more knowledge level issues to deal with then stop expanding a task"
Design Step Proposal	Propose step	Propose1 PSM	Role expansion	Software Eng.	"If it is possible, reflect changes in semantics of role by changes in labels"

Table 5: Examples of design principles and their indices.

Table 5 gives examples of the design principles which we have induced from our own experience in KBS design and from standard software design practice. It is not our aim to provide a complete or even a consistent list of such principles—indeed we see the ongoing practice of KE design (within particular methodologies) as a process of adding to these design principles. Any principle can be violated if there is a good reason and some of these principles are mutually inconsistent.

The Design Principles act as ways of transforming the target model (or at least shifting the design focus) and updating the design rationale. Seen as elements in the arguments

contained in the design rationale, they provide some kind of 'standard reasons' for typical design decisions. These could be very useful for naive KBS designers—for instance our knowledge engineering students—because they can learn the principles behind good design. They can also have some utility for the experts at the very least by providing a standard reference model for recording design decisions. In short, they fulfil Van de Velde's call (1994b: 727) for a "repository of knowledge engineering experience and practice that is useful and usable to the wider community of knowledge engineers" since they represent an accumulating store of "best practices" which partakes in the argumentational processes of design. The hierarchical library of principles can be seen both as storing the distilled essence of ongoing design activities and as a means for ensuring that good designs are produced.

In Table 5 we have made a distinction between the levels of General Model and General Argument. These serve to distinguish between principles which apply to design model components (other than the design rationale) and those which apply only (or mainly) to the design rationale itself. The Propose1 problem solving method introduced in the column labelled 'Attached to' is derived from the example given in section 5.

4.4. Design Steps—Operational Design Principles

The process of knowledge-level design can be summarised in the following sequence of steps. The reader should note that the steps are presented as an algorithm for convenience and succinctness; while the design process may be computable in restricted domains, an algorithmic solution is not readily available either for innovative design or design decision-making. This, of course, is why we need to argue about the decisions we make even in a fairly constrained domain like KBS design. Moreover, again for the sake of brevity, we have not included any verification and revision steps.

1. Select current task (e.g. VT design problem).
2. Classify task (e.g. VT is a parametric design problem). This means that the specific task in hand 'inherits' the characterisation in terms of inputs and output roles, constraints, goals, and domain requirements from the chosen class.
3. Map task roles to domain knowledge and instantiate domain requirements in the given domain. This step might require acquiring the relevant domain knowledge structures if they have not yet been acquired.

4a. Select suitable PSM from library - e.g. from (Breuker & Van De Velde, 1993) or (Le Roux at el., 1993). The PSM doesn't need to be fully specified. For instance, one could decide to tackle VT by using Propose&Revise. However, as already mentioned, Propose&Revise is a label which we use to characterise an entire class of problem solvers exhibiting diverse control regimes and problem solving

architectures. Alternatively, one can select a fully specified, primitive problem solving method, which provides a 'full' solution to the task in hand.

4b. If there is no PSM in the library which is applicable then develop a model of a new one by carrying out a *task analysis* of the problem solving behaviour of an expert in the given domain. This can be done by using techniques such as protocol analysis or thinking aloud.

5. Once a PSM has been selected, then it is applied to the task in hand. This process could be seen as that of defining a mapping between the input and output roles of the PSM and those of the task. This view characterises the task-to-method mapping as analogous to unification between formal and actual parameters in programming languages. However, in many cases things are not so simple. For instance let's consider the case in which we want to tackle the VT design problem by means of Propose&Revise. As shown by the definitions in Figures 2 and 4, parametric design problems have an I/O specification which is different from that which characterises Propose&Revise problem solvers. In particular, there is no provision for model extension procedures and fixes in the specification of parametric design tasks. Therefore when applying a problem solving method we need to map its roles and domain requirements to the knowledge structures in the domain, in the same way we do when mapping a task specification to a domain. Moreover, it is also crucial to map the solution role(s) of the task to some output role of the PSM.

6. If the PSM just applied is a primitive one then we are done with the current task and we go to step 8.

7. If the PSM is a structured one, then the tasks it introduces are added to the list of 'pending' tasks and we go to step 8.

8. If there are still tasks to tackle we go back to step 1. Otherwise we go to step 9.

9. If all tasks have been solved we check whether all inter-task control relations have been specified. If they have not we specify them thus completing the architecture of our problem solving model.

4.5. OCML-DDL—Mini-Manual for language

The following are the main elements in the OCML-DDL design decision language. We provide a BNF grammar for each construct, an informal description, and an example.

Design Decision

```
<DesignDecision>        ::=  (DesignDecision
                                 :design-context <DesignContext>
                                 :decided text
                                 :justification text
                                 :grounds a-list
                                 :alternative-proposals a-list
```

:outcome semi-formal-text
:object <Object>
:decided-by text
:date text
:address text)

A design decision primarily records what has been decided and the justification (which frequently either refers to a pre-existing design principle or represents a new principle). The decision also records the outcome of the process: i.e. the change in the target model and the particular object about which the decision is made. Provision is also made for various annotations. Since decisions are made in particular contexts a design decision may record the current design context (see below). Since, in many cases, alternative decisions might have been made the designer can record the alternative proposals (each of which has its own argumentation—see below).

In relation to the object of the decision, the question arises as to how best attach justifications. In some cases an obvious candidate is available. In others the decision is about a chunk (several model components) or about the model as a whole.

It would be possible to produce a typology of design decisions based on either the current design step or the object being decided about but this is redundant here since both are recorded. We can think of two main types depending on whether or not any choice had to be made between design proposals. An example of a design decision follows:

```
(DesignDecision
      :decided "Select Propose&Revise method"
      :justification "Suitable for VT task"
      :outcome new-decision
      :object Initial-model
      :decided-by Arthur Stutt
      :date 8-93
      :address HCRL)
```

Design Context

<DesignContext> ::= (DesignContext
:context-name a-name
:focus <Object>
:goal <Goal>
:state-of-design <a-state>
:operative-constraints <Domain/Spec/Design>
:agenda a-list)

The design context provides a record of important aspects of the design being created during the design process. In particular it records shifts in the focus and goal of the design process, the constraints (whether deriving from the nature of the domain, the

requirements specification, the expert's problem solving model, or the state of the design so far) which are operational, and an agenda which contains undischarged design tasks. As such the design context is dynamic and not part of the design history. Below we show an example of a design context.

```
(DesignContext
      :focus Propose
      :goal "Expand sub-task Propose"
      :operative-constraints
           (psmP&R requirements-spec domain-ontology)
      :agenda ("Expand sub-task Evaluate" "Expand sub-task Revise"))
```

Design Decision Argument

```
<DesignDecisionArgument>  ::= (DesignDecisionArgument
                                :design-context <DesignContext>
                                :proposal <Claim>                    ;;claim
                                :pro <Args>*                         ;;grounds
                                :contra <Args>*                      ;;possible rebuttal
                                :specific-principle <Principle>      ;;warrant
                                :principles-involved <Principles>    ;;backing
                                :inference-type <Deduction/Analogy/etc>
                                :supports <Arg>
                                :attacks <Arg>
                                :restates <Arg>
                                :participant text)
```

The design decision argument is the structure which records the extended Toulmin argument. As well as the design context, the pro and cons arguments, the principles involved and the inference type (deduction, analogy, induction, abduction), the structure records the relationship of a particular instance to other design decision arguments as either supports, attacks, or restates. This extension to the basic Toulmin model means that it is capable of recording complex multi-participant arguments while at the same time providing the means to give details of each micro-argument.

```
(DesignDecisionArgument
      :proposal "Use Propose1"
      :pro "Yost document page 2"
      :specific-principle "Follow specification"
      :principles-involved #vital-principles
      :attacks "Use ProposeAll")
```

Design Constraint

```
<DesignConstraint>        ::=  (DesignConstraint
                                   :design-constraint-name
                                   :constraint
                                   :design-context
                                   :constraint-type
                                   :object-constrained
                                   :fix)
```

A design constraint is of four main types depending on whether they are derived from: the domain ontology, the expert's PSM, the requirement's specification or the state of the target model so far (the 'logic' of the design). In addition there are high level constraints provided by the structure of the argument so far (the 'logic' of the augmented design model). In addition this structure may record the design context, the object (or model component) constrained and the means for fixing the constraint violation.

```
(DesignConstraint
      :design-constraint-name #constraint1
      :constraint "Repeated tasks must have a stopping condition"
      :constraint-type target:general-constraints-on-tasks
      :fix "insert stopping condition")
```

Design Fault

<DesignFault> ::= (DesignFault
:design-fault-name a-name
:fault text
:object <Object>
:design-context
:operative-constraint-violated)

The design fault records the fault, its object and the constraint violated.

```
(DesignFault
      :fault "redundant role"
      :object (ROLE: 'Output')
      :operative-constraint-violated "Remove redundancy")
```

Design Fix

<DesignFix> ::= (DesignFix
:design-fix-name a-name
:fault text
:object <Object>
:design-context
:how-to-fix text)

The design fix records details of how to fix a design fault.

```
(DesignFix
      :fault "redundant role"
      :object (ROLE: 'Output')
      :how-to-fix
            "In order to fix the redundant application of a task
            impose conditions which need to be satisfied before the
            task is invoked")
```

5. AN EXAMPLE—OCML-DDL IN USE

The following example is drawn from the discussion of the VITAL solution to the VT lift configuration problem which is given in full in Motta, et al. (1994). The main area

of difficulty in lift design is in ensuring that there are no conflicts between the types or dimensions of two different components. If conflicts do arise there must be some means of fixing these. The VT problem is thus one in which the system must cycle through phases of assignment of values to components and dimensions, checking if constraints are violated, and fixing these violations until either it fails or a satisfactory solution is reached. The lift configuration problem naturally lends itself to the propose-evaluate-revise scheme suggested for lift configuration in Marcus et al. (1988). In what follows we present a series of design steps reconstructed from our solution. We will assume that the decision to use the VITAL methodology has been selected. For each step we provide: (a) a discussion of the difficulties faced, the alternatives and the solution chosen, and (b) a OCML-DDL representation of part of the design decision. For brevity, we only include the OCML-DDL notations for enough of the design decisions to show what is at issue in the design of the VT system and to illustrate the utility of the design decision approach.

Design step 1

Once the VT problem has been characterised as a parametric design problem, all roles have been mapped to domain knowledge structures, and the domain requirements imposed by the specification of parametric design tasks have been satisfied, the next step in the modelling process is to select an initial PSM from a library or, alternatively, to define one. In this case the choice is relatively simple. As pointed out in (Chandrasekaran et al., 1990) all design methods can be seen as variations of the propose-evaluate-revise schema, whose I/O specification is shown below. (The prefix M: indicates that the node is a PSM. We use the prefix T: for tasks.)

Figure 9: The I/O specification for the class of Propose&Revise PSMs

The relevant Design Decisions are straightforward:

```
(DesignDecision
      :decided "Select initial PSM"
      :justification "Step 4a in VITAL design steps"
      :outcome new-decision
      :object Initial-PSM
      :decided-by Arthur Stutt
      :date 8-93
      :address HCRL)
```

```
(DesignDecision
      :decided "Select P&R PSM"
      :justification "Most generic PSM for parametric design"
      :outcome Initial-PSM=P&R
      :object Initial-PSM)
```

Design step 2

The second step involves the instantiation of the P&R PSM: i.e. the roles of the PSM are mapped to roles in the domain ontology (or alternatively the relevant knowledge structures are acquired). The solution described in (Motta et al., 1994) was based on the Yost specification of the VT problem (Yost, 1992) which provides a description of the VT domain which maps very easily to a Propose&Revise schema. In order to make this design process more interesting we assume that the only information available to us is instead the Ontolingua specification of the VT KB (VT-Onto), which is described in (Gruber et al., 1994). This specification describes VT as a parametric design task, in terms of components, parameters, and constraints - see figure 10.

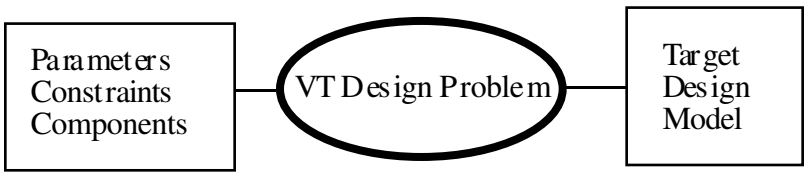


Figure 10: Functional view of the Ontolingua VT KB.

It is obvious that the mapping between the functional specification of the Propose&Revise class of methods shown in figure 9 and the structures which are part of VT-Onto is not trivial. VT-Onto doesn't contain any knowledge about fixes, nor makes explicit the distinction between propose steps and constraints. Therefore performing this mapping will require additional knowledge acquisition, for instance an in-depth analysis of the knowledge encoded in VT-Onto. A design decision generated at this stage is shown below, specifying that knowledge about fixes needs to be acquired in order to satisfy one of the input requirements of Propose&Revise.

```
(DesignDecision
      :decided "Acquire knowledge about fixes"
      :justification
            "Needed by P&R PSM"
      :decision-step 2
      :object Initial-PSM-Roles)
```

Design step 3

The next step is to select a new focus. The embryonic PSM for P&R has the following internal structure:

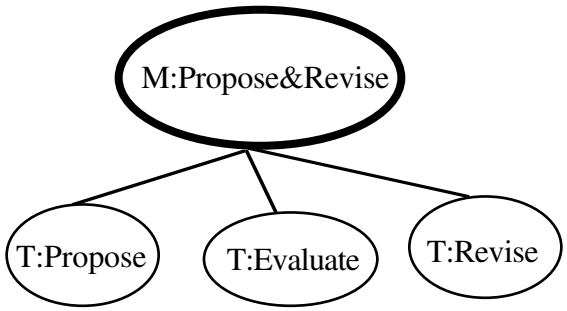


Figure 11: The task structure of the Propose&Revise PSM

In general unless there is some contra-indication we proceed in a depth first manner from left to right. Thus the next focus is the Propose task provided by the P&R PSM.

```
(DesignDecision
      :decided "Focus on the Propose sub-task"
      :justification
            "In general decomposition is depth-first left-right"
            (see Table 1)
      :outcome focus(Propose)
      :decision-step 3
      :object Propose)
```

This design decision also has the effect of altering the design context. The context keeps track of the current focus and the operative constraints. In addition the design context includes an agenda of design goals to be pursued. The design will be incomplete if there are pending design goals.

```
(DesignContext
      :focus Propose
      :goal "Expand sub-task Propose"
      :operative-constraints
           (psmP&R requirements-spec analysis-model)
      :agenda ("Expand sub-task Evaluate" "Expand sub-task Revise"))
```

Design step 4

We now need to expand the Propose sub-task. The alternative steps here are selecting a PSM, selecting a partial PSM, or creating a partial PSM. In this case PSM selection is relatively easy since the PSM for P&R has two default methods attached. In other cases we would have to iterate through a process of selection, mapping and checking until a suitable (partial) PSM is found.

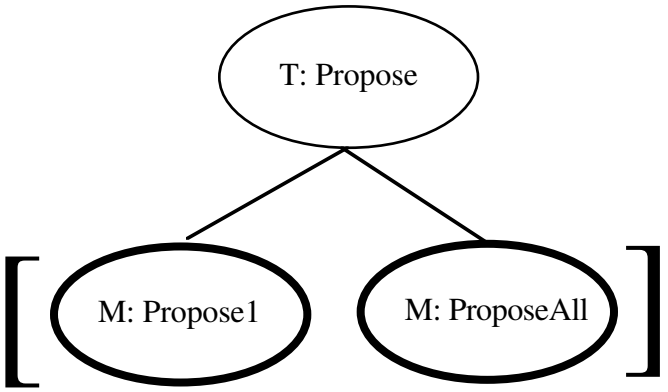


Figure 12: The possible methods for the Propose task

The choice has to be made at this point between two conflicting methods for expanding the Propose task. These are derived from the domain description given in Yost (1992) and represent two different strategies for propose-evaluate-revise:

Complete-model-then-revise (CMR): i.e. during the propose phase, the values of all parameters in VT are computed, before any check is made to see if there are constraint violations.

Extend-design-then-revise (EMR): i.e. the model is extended piecemeal and checked for constraint violations after each model extension.

The CMR approach has one important advantage over EMR due to the fact that, during the revision stage, the CMR method can exploit the interdependencies between constraints and between constraints and fixes, in order to select and apply those fixes which minimise the number of remaining constraint violations. As discussed in detail in (Minton et al., 1992) this strategy can produce significant improvements in the performance of the problem-solver, especially when the number of interdependencies between constraints is high. For this reason, we make the decision to use the CMR method, i.e. to select ProposeAll rather than Propose1.

```
(DesignDecision
      :decided "Use the ProposeAll method"
      :justification
            "ProposeAll implements the CMR strategy, which allows us
            to exploit the interdependencies between constraints and
            fixes"
      :alternative-proposals ("Use Propose1" "Use ProposeAll")
(DesignDecisionArgument
      :proposal "Use Propose1"
      :pro "Yost page 2"
      :specific-principle "Follow specification"
      :principles-involved #vital-principles)
(DesignDecisionArgument
      :justification-name #arg8
```

```
:proposal "Use ProposeAll"
:pro "Yost page 5"
:specific-principle "Follow specification"
:principles-involved #vital-principles)
```

Design step 5

As with the second design step we must now instantiate the ProposeAll method. Since no new roles are introduced this is trivial.

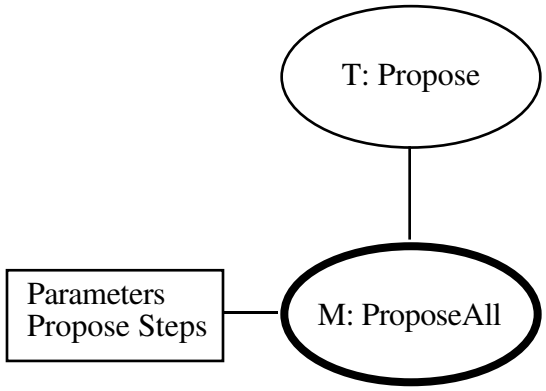


Figure 13: The ProposeAll method is chosen

Design step 6

We continue the expansion of the Propose task by focusing on the sub-tasks introduced by the instantiated method ProposeAll. In this case ProposeAll is comprised of iterative invocations of the Propose1 task with a stopping condition (our first control introduction).

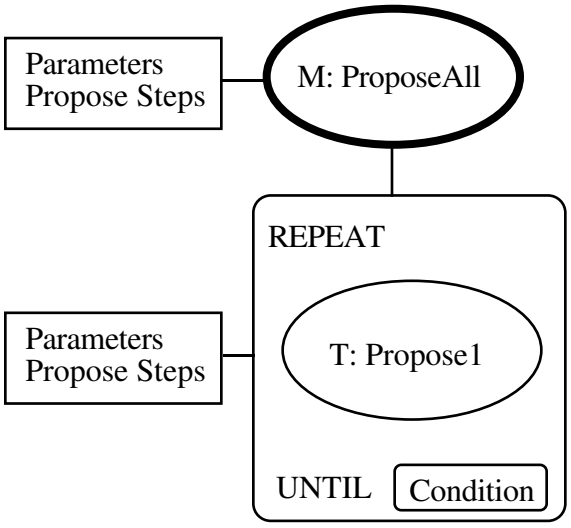


Figure 14: The task structure of ProposeAll

There is no choice here but to focus on Propose1 and continue expansion. The PSM for ProposeAll has a default method attached to the task Propose1 which we select. Again, as with step 5 the instantiation of Propose1 is trivial—there are no new roles.

Design step 7

The default Propose1 method is composed of three sub-tasks: Select Parameter Select Propose Step and Apply Propose Step (i.e. use the role Propose Step to assign a value to the parameter).

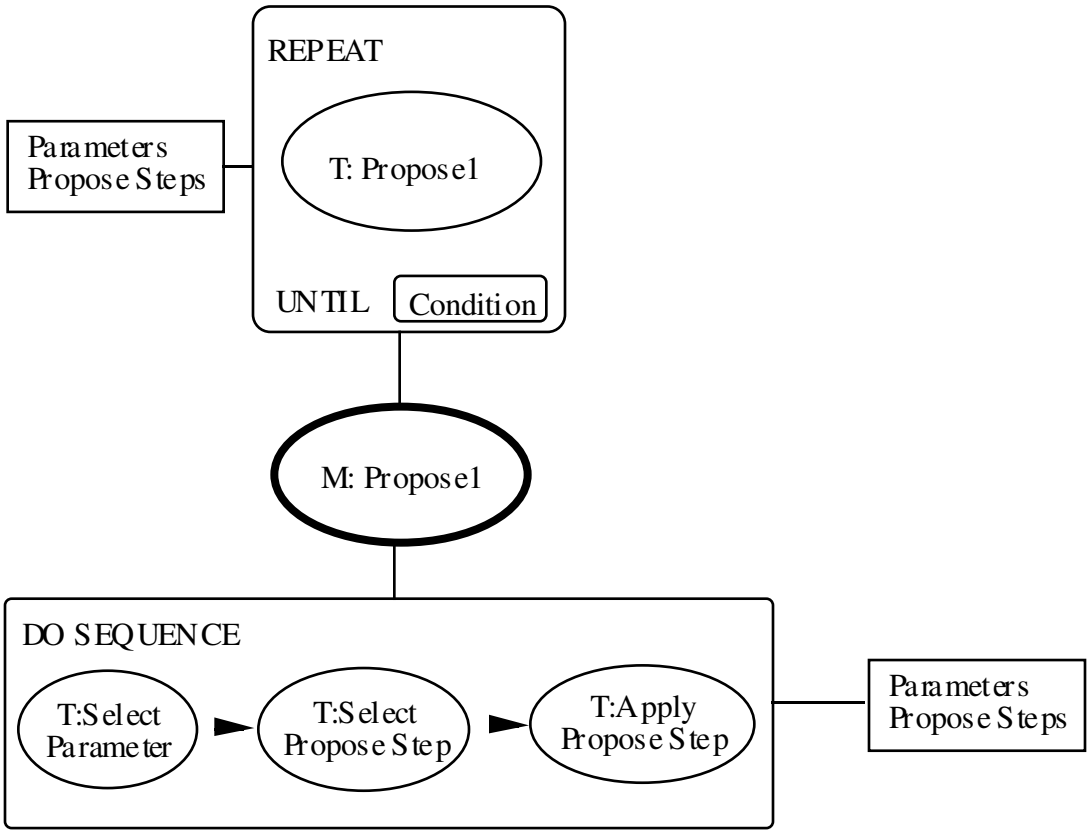


Figure 15: The task structure of Propose1

Because we have primitive methods for each task introduced by the Propose1 method we can stop the task decomposition process here. The figure below shows a definition of a simple method for selecting parameters, taken from (Motta, In Progress). The method specifies that the parameter to be selected is one which is unassigned, and whose value is computed from parameters which have all been assigned.

```
(define-instance DEFAULT-SELECT-PARAMETER-METHOD (primitive-ps-
method)
   "The default method for task select-parameter just gets one of the
    unassigned parameters"
   :slots ((suitable-for select-parameter)
           (requires-domain-relation depends-on)
           (body  (lambda (?task)
                    (if (exists ?param
                                (and (unassigned ?param ?task)
                                     (forall ?dparam
                                             (=> (depends-on ?param
                                                             ?dparam)
                                                 (assigned ?dparam
                                                           ?task)))))
                        ?param))))
```

Figure 16: Simple parameter selection method

Design step 8

Since there is no need for revision at this stage the designer now looks for a new focus. Since the design agenda has pending design goals ("Expand sub-task Evaluate" "Expand sub-task Revise") the designer chooses the former.

An example later design step

For brevity we select a later step which illustrates the use of OCML-DDL. We are now focusing on the expansion of the Revise sub-task, and, in particular, on the task for fixing constraint violations by means of heuristic fixes. We know from the domain data that there are different types of fix and that these can be combined in a number of ways. As a result of this we need to introduce new tasks (and their associated roles) for generating *fix combinations*, for selecting a particular combination, for applying this, for checking to see if new constraint violations are created, for checking that the current constraint violation is fixed and for undoing changes if necessary. As an example of the design process involved (and its tortured argumentation) we discuss the problem of deciding what to do when all fix combinations for a particular constraint violations have been tried and they have all failed. There are two alternatives here. We could decide we cannot fix the current constraint violation, halt the design process and return failure. Alternatively, we could record the current constraint violation (say CV1) as 'unsolved' and move on to fix another one. If this approach is taken, the system can attempt to fix CV1 after other violations have been dealt with. This decision means that we must extensively restructure our design model to include tasks for checking for unsolved constraint violations (and the appropriate roles).

```
(DesignDecision
      :decided "Add new task and roles to incorporate 'unsolved' CVs"
      :justification "Improvements to the competence of the problem
                      solver should be carried out even if
```

```
                this means extensive revision of the
                design model is needed"
    :grounds ("Unfulfilled constraints exist")
    :alternative-proposals ("Return failure"))
```

6. CONCLUSION—OCML-DDL IMPROVES THE VITAL DESIGN METHODOLOGY

We believe that OCML-DDL offers a means for capturing design decisions which is natural, succinct and expressive, and which overcomes the possibility that the argumentation process detracts from the design process by:

(a) ensuring that the user is not tied to a particular sequence of steps and can ignore argumentation activities until s/he needs them;

(b) providing a design language which bases the argumentation behind design decisions on design principles which operate at different levels (and are therefore more likely to be meaningful to designers).

The main strength of OCML-DDL is that design decisions are made in the context of a set of hierarchically organised design principles which express a clearly thought out design method. Work is continuing on this body of principles but, since it cannot hope to be exhaustive, we intend to develop a tool for extracting design principles from design rationales. In addition, new strategies—other than stepwise PSM expansion—will be investigated and added to the design principles library.

The second outcome of our approach is that, in providing a language for recording design decisions as well as libraries of model components and structured design principles, OCML-DDL serves as a modeling repository (in Van de Velde's sense) which indicates to future knowledge engineers how model components have been combined by actual designers (the design rationales) and how they may best be combined (design principles).

Nonetheless, OCML-DDL remains open to criticism from researchers such as Buckingham Shum and Hammond (1994) when they question the usability and utility of design rationales. In order to respond to such criticism we need to do further work, firstly, to collect and analyse protocols of designers making decisions and to extend (or amend) OCML-DDL as necessary, and, secondly, to incorporate OCML-DDL into a suite of support tools (such as editors and browsers).

In the end then, the proof of the design method pudding is not in the arguments given above but in its acceptability to designers (whether professional or student). This may be seen as a paradox given our foregrounding of argumentation.

REFERENCES

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

Breuker, J. A. and Van de Velde, W. (1993). The CommonKADS Library. ESPRIT Project P5248 Deliverable M2, University of Amsterdam and Free University of Bruxelles.

Buckingham Shum, S. and Hammond, N. Argumentation-based design rationale: what use at what cost? International Journal of Human-Computer Studies, 40(4), 603-652, April 1994.

Chandrasekaran, B. Design Problem Solving: A Task Analysis. AI Magazine. 11(4), 1990. 59-71.

Chandrasekaran, B., Johnson, T.R. and Smith, J.W. Task-Structure Analysis for Knowledge Modelling. Communications of the ACM, 35(9), Sept. 1992. 124-137.

Conklin, J. and Begeman, M.L. gIBIS: A Hypertext Tool for Exploratory Policy Discussion. ACM Trans. on Office Information Systems, 6(4), 1988. 303-331.

Domingue, J.B., Motta, E., and Watt, S. The emerging VITAL Workbench. In N. Aussenac et al (eds) Knowledge Acquisition for Knowledge-Based Systems: Proceedings of the 1993 European Knowledge Acquisition Workshop. Springer-Verlag. 1993.

Eisenberg, M. and Fischer, G. Programmable Design Environments: Integrating End-User Programming with Domain-Oriented Assistance. Proc. CHI'94 (Human Factors in Computing Systems). Boston, Mass., ACM, 1994. 431-7.

Fischer, G., Girgensohn, A., Nakakoji, K. and Redmiles, D. Supporting Software designers with Integrated Domain-Oriented design Environments. IEEE Trans. on Software Engineering. 18(6), June 1992. 511-52.

Fischer, G., Lemke, A.C., McCall, R. and Morch, A. I. Making Argumentation Serve Design. Human-Computer Interaction. 6, 1991. 393-419.

Fischer, G., McCall, R. and Morch, A. Design Environments for Constructive and Argumentative Design. Proceedings of the ACM conference on Human Factors in Computing Systems (CHI-89), May 1989. 269-275.

Genesereth, M.R., Fikes, R.E.(1992). Knowledge Interchange Format, Reference Manual, Version 3.0, *Technical Report*, Computer Science Department, Stanford University, 1992.

Gennari, J. H., Tu, S. W., Rothenfluh, T. E., Musen, M. A. Mapping Domains to Methods in Support of Reuse. Proceedings of the 8th Banff Knowledge Acquisition Workshop, Banff, Canada, 1994.

Gruber, T. R. A Translation Approach to Portable Ontology Specifications. *Knowledge Acquisition*, *5(2)*, 1993.

Gruber, T. R., Runkel, J. T., and Olsen, G. R. VT Domain Ontology. Available by FTP from ksl.stanford.edu/pub/knowledge-sharing/ontologies/. Stanford: Knowledge Systems Lab., Computer Science Dept. Stanford University, 1994.

Gruber, T.R. and Russell, D.M. Design Knowledge and Design Rationale: A Framework for Representation, Capture, and Use. Technical Report KSL 90-45. Stanford: Knowledge Systems Lab., Computer Science Dept. Stanford University, 1991.

Jarczyk, A.P.J., Löffler, P. and Shipman, F.M. Design Rationale for Software Engineering: A Survey. Proceedings of the 25th Hawaii Int. Conf. on System Sciences, Jan 1992. 577-586.

Jonker, W. and Motta, E. (1993). Functional design activity using $K_{BS}SF$ and VITAL-OCML. VITAL Project Report DD221 (Part III), PTT Research, Groningen, March 1993.

Jonker, W. and Spee, J. W. Yet another formalization of KADS conceptual models. In Wetter et al. (Eds), *Current Developments in Knowledge Acquisition*. Lecture Notes in Artificial Intelligence, 599, Springer-Verlag, Berlin, 1992.

Landes, D. and Studer, R. The Design Process in MIKE. Proceedings of the 8th Banff Knowledge Acquisition Workshop, Banff, Canada, Vol 2, 1994. 33-1 - 33-20.

Le Roux B., O'Hara K., Outtandy S., Shadbolt N., Laublet P. and Motta E. (1993). The VITAL Library of Knowledge Modelling. *VITAL document DD215.* Project VITAL (ESPRIT II Project P5365). Onera.

Lee, J. and Lai, K.-Y. What's in Design rationale? Human-Computer Interaction, 6, 1991. 251-280.

Lemke, A.C. FRAMER-HYPERTEXT: An Active Issue-Based Hypertext System. Technical Report. Dept. of Computer Science, University of Colorado, Boulder, 1990.

Marcus S., Stout J. and McDermott J. VT: an expert elevator designer that uses knowledge-based backtracking. AI Magazine, 9(1), 1988. 95-112.

McCall, R.J. PHI: A Conceptual Foundation for Design Hypermedia. Design Studies. 12(1), 1991. 30-41.

McCall, R., Fischer, G. and Morch, A. Supporting Reflection-in-Action in the Janus Design Environment. In: eds. McCullough et al. The Electronic Design Studio. Cambridge, MA: MIT Press, 1990. 247-259.

Minton S., Johnson M.D., Philips A.B. and Laird P. (1992). Minimising conflicts: a heuristic repair method for constraint satisfaction and scheduling problems. *Artificial Intelligence 58*. (1992). pp. 161–205

Motta, E. Representing Tasks and Problem Solving Methods in Ontolingua. Work in progress, Human Cognition Research Laboratory, The Open University, UK.

Motta, E., O'Hara, K., Shadbolt, N., Stutt, A. and Zdrahal, Z. A VITAL Solution to the Sisyphus II Elevator Design Problem. Proceedings of the 8th Banff KA for KBS Workshop, Vol 3, 1994.

Newell, A. The Knowledge Level. *Artificial Intelligence*, 18(1), 1982. 87-127.

Puerta, A. R., Egar, J. W., Tu, S. W., and Musen, M. A. A multiple-method knowledge-acquisition shell for the automatic generation of knowledge-acquisition tools. Knowledge Acquisition 4(2), 1992, 171-196.

Ramesh, B. and Dhar, V. Supporting Systems Development by Capturing Deliberations During Requirements Engineering. IEEE Trans. on Software Engineering. 18(6), June 1992. 498-510.

Ramesh, B. and Dhar, V. Representing and Maintaining Process Knowledge for Large-Scale Systems Development. IEEE Expert, 9(2), April 1994. 54-59.

Rathke, C. and Negele, A. KONEX+: A Knowledge-Based Design Environment. Proc. of the Eleventh Int. Workshop on Expert Systems and Their Applications. General Conference: Tools, Techniques and Methods, Vol 1, Avignon, France, 1991. 147-159.

Rich C., and Waters, R. C. *The Programmer's Apprentice*, ACM Press, New York, 1990.

Runkel, J. T., Birmingham, W. B., Balkany, A. Separation of Knowledge: a Key to Reusability. Proceedings of the 8th Banff Knowledge Acquisition Workshop, Banff, Canada, 1994.

Schreiber, G. Operationalizing Models of Expertise. In: Schreiber, Wielinga and Breuker, 1993. 119-149.

Schreiber, G. Reference Model for the VT Domain. Unpublished draft memo, 1994.

Schreiber, G., Bredeweg, B., Davoodi, M. and Wielinga, B. Towards a Design Methodology for KBS. ESPRIT Project P1098 Deliverable D8 UvA/STC-B2-PR-001 (VF Memo 97). University of Amsterdam, 1987.

Schreiber, G., Wielinga, B. and Breuker, J. Eds. KADS: A Principled Approach to Knowledge-Based System Development. London: Academic Press, 1993.

Simon, H. A. The Science of Design: Creating the Artificial. Ch 5, The Sciences of the Artificial. Cambridge:MIT Press, 1981. 128-159.

Steels, L. Components of Expertise. AI Magazine. 11(2), 1990. 29-49.

Steels, L. Reusability and configuration of applications by non-programmers. VUB AI Lab Memo 92-4. VUB AI Lab, Brussels, 1992.

Stutt, A. Argument in the humanities: a knowledge-based approach. Unpublished PhD Thesis. 1989. (Available as Technical Report No. 49. HCRL, The Open University, 1989.)

Tansley, D.S.W. and Hayball, C.C. Knowledge-Based Systems Analysis and design: A KADS Developer's Handbook. New York: Prentice Hall, 1993.

Terpstra, P. and Schrooten, R. CommonKADS Specific Design Decisions and their Notation. Technical Report ESPRIT Project P5248 KADS-II/M7/UvA/043/1.1. University of Amsterdam, 1994.

Toulmin, S. The Uses of Argument. Cambridge: Cambridge University Press, 1958.

Toulmin, S., Rieke, R. and Janik, A. An Introduction to Reasoning. New York: Macmillan Publishing Co, 1979.

Van de Velde, W. Issues in Knowledge Level Modelling. Proceedings of the 8th Banff KA for KBS Workshop, Vol 2, 1994a. 38-1 - 38-11.

Van de Velde, W. A Constructivist View on Knowledge Engineering. Proc. 11th European Conf. on AI. Ed. Cohn, A. London: Wiley and Sons, 1994b. 727-731.

Van de Velde, W., Duursma, C. and Schreiber, G. Design Model and Process. Technical Report ESPRIT Project P5248 KADS-II/M7/VUB/RR/064/1.1. Free University of Brussels, 1993.

Wielinga B. J., Schreiber A.Th. and Breuker J. KADS: A Modelling Approach to Knowledge Engineering. In *Knowledge Acquisition* 4(1) pp.5-53, 1992a.

Wielinga, B., Van de Velde, W., Schreiber, G. and Akkermans, H. The CommonKADS Framework for Knowledge Modelling. Proceedings of the 7th Banff KA for KBS Workshop, Vol 2, 1992b.

Yost, G. R. Configuring Elevator Systems. DEC Project Report. 1992.

Zdrahal, Z and Motta, E. An In-depth Study of Propose & Revise Problem Solving. Proceedings of the Japanese Knowledge Acquisition Workshop, JKAW '94, Hatoyama, Japan, 1994.

APPENDIX—THE VITAL EXTENDED OCML (VITAL-OCML-DDL)

```
<DesignDecision>            ::=  (DesignDecision
                                     :design-context <DesignContext>
                                     :decided text
                                     :justification text
                                     :grounds a-list
                                     :alternative-proposals a-list
                                     :outcome semi-formal-text
                                     :object <Object>
                                     :decided-by text
                                     :date text
                                     :address text)

<DesignContext>             ::=  (DesignContext
                                     :context-name a-name
                                     :focus <Object>
                                     :goal <Goal>
                                     :state-of-design <a-state>
                                     :operative-constraints <Domain/Spec/Design>
                                     :agenda a-list)

<DesignDecisionArgument>    ::= (DesignDecisionArgument
                                    :design-context <DesignContext>
                                    :proposal <Claim>                  ;;claim
                                    :pro <Args>*                       ;;grounds
                                    :contra <Args>*                    ;;possible rebuttal
                                    :specific-principle <Principle>    ;;warrant
                                    :principles-involved <Principles>  ;;backing
                                    :inference-type <Deduction/Analogy/etc>
                                    :supports <Arg>
                                    :attacks <Arg>
                                    :restates <Arg>
                                    :participant text)

<DesignConstraint>          ::=  (DesignConstraint
                                     :design-constraint-name
                                     :constraint
                                     :design-context
                                     :constraint-type
                                     :object-constrained
                                     :fix)

<DesignFault>               ::=  (DesignFault
                                     :design-fault-name a-name
                                     :fault text
                                     :object <Object>
```

```
                                    :design-context
                                    :operative-constraint-violated)

<DesignFix>              ::=  (DesignFix
                                    :design-fix-name a-name
                                    :fault text
                                    :object <Object>
                                    :design-context
                                    :how-to-fix text)
```