Recording the design decisions of knowledge engineers to facilitate re-use of design models.

Arthur Stutt and Enrico Motta

Knowledge Media Institute,
The Open University,
Walton Hall,
Milton Keynes MK7 6AA,
United Kingdom
a.stutt, e.motta@open.ac.uk

Abstract. In this paper we focus on the process of constructing reusable *knowledge level* models by augmenting an explicit process model of KBS design with a means of recording the argumentation about design decisions. Our method includes a set of design principles and an expressive design language for representing design components (such as tasks and roles) with an extension for *decision descriptions.* We present a concrete example of the application of our design methodology which illustrates how our design language can be used to produce a communicable and re-usable design model.

1. INTRODUCTION

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, they "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.

In this paper, then, we focus on the process of constructing reusable *knowledge level* design models. By knowledge level model we mean the structured, solution-oriented models identified by Van de Velde (1994a: 38-4). By design model we mean both the representation of design components such as tasks, roles and methods (see next section), and the design rationale. Re-use is aided by augmenting design models in this way 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 this involves—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 components, 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 including 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).

2. METHODOLOGICAL BACKGROUND

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 of design components.

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 specification describes what needs to be achieved. For instance in the VT problem (Yost, 1994) the task is to assign values to the design parameter in a way which satisfies all given constraints. VT 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 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 the information about parts of an elevator and associated constraints which is required to configure an elevator.

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. This definition is expressed in OCML (Motta, 1995), our own language for knowledge-level KBS modelling.

The expression in Figure 1 describes the task of parametric design as taking a set of constraints, components, and parameters as input and producing a design model[1]. The goal is to ensure that each parameter has a value and that each constraint is satisfied. In addition the definition also

---

[1] The reader should note that the design model discussed here is the design model *produced by* a system for automating configuration design. The design model discussed in the rest of the paper is the design model *of* such a system.

assumes that a specific parametric-design domain will provide the definition of a relation, HAS-VALUE, modelling the assignment of values to parameters. Figure 2 shows an example of mapping a task role to a domain. In the example, the input role 'parameters' of the VT design task has been mapped to the set of parameters specified in an OCML-based model of the VT domain (Motta et al., 1995).

```
(def-task-type parametric-design (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 organized around components.
    The specification of a parametric design task requires that the
    domain relation HAS-VALUE (?parameter ?value) is specified in the
    theory"
   ((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)
                            (satisfied ?c)))))
    (requires-domain-relation has-value)))
```

Figure 1: A sample task specification: Parametric Design

```
(def-axiom reference-for-parameters-role
    "This axiom specifies that the role `parameters'
     in the definition of the VT parametric design task maps
     to the set of all parameter slots in the vt domain ontology."
   := (domain-reference
      parameters
      vt-design-problem
      (setofall ?p
            (exists ?c
               (and
                  (vt-component ?c)
                  (has-parameter ?c ?p))))))
```

Figure 2: 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 to 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.

Figure 3 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 *model extension 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 that problem solving methods are selected to cover each task.

```
(def-psm Propose&Revise ?method
   "The Propose & Revise class of PSMs 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 revision strategies
   associated with constraints and returns a target model
   defined as a consistent assignment of values to parameters"
   ((introduces-subtask Propose Evaluate Revise)
    (requires-domain-relation model-complete?)
    (subtask-control-regime
      :default-value
         (loop
            (do Propose Evaluate Revise)
                until (model-complete?
                        (domain-reference target-model
                                          ?method))))
    (has-input-role Model-Extension-Procedures Parameters
         Constraints Revision-Strategies)
    (has-output-role Target-Model)))
```

Figure 3: A coarse-grained specification of the Propose&Revise method

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. 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).

3. DESIGN AND ARGUMENTATION

The propose-evaluate-revise schema used in discussing design as a generic task (Chandrasekaran, 1990) makes explicit its non-monotonic character. This schema operates at many different levels. As well as reflecting the frequent revisions in design representations needed before the final design is produced, it also operates at a higher level as the designer makes decisions between competing options, checks these (perhaps against the reasoning of other designers) and makes revisions as necessary, before changing the target design model. We normally refer to this higher level process as argumentation. Argumentation (at least with one's self) can thus be seen to form part of the cognitive processes involved in design. Since we are interested in capturing the history of decision-making as well as the final non-monotonic decisions reached, the design rationale must be monotonic—i.e., it should record all the reasoning about the design.

3.1. Recording the arguments behind the design—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, with supporting chains of justification, attached to design components or the design as a whole. At each step in the design process the design components are transformed and/or the (relevant node in the) design rationale is updated.

The literature (Gruber and Russell, 1991; Fischer et al., 1991; Ramesh and Dhar, 1992; 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 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. Models of Argumentation

There are several models of argumentation—the theory behind the design rationale—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 ensures 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 theories.

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) 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). 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.

3.3. Approaches to using argumentation in designing

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 a generic hypertext based on the IBIS approach is utilised as a means of articulating the design rationale—e.g., gIBIS (Conklin and Begeman, 1988).

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 systems such as JANUS (McCall, Fischer and Morch, 1990)—which integrates the CRACK critiquing design system with support for argumentation—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 is the unselfconscious deployment of design expertise when things are going well, while the latter is needed when there is some 'breakdown' between the designer's expectations and what happens when a design step is made. 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 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 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

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.

While much is gained by the use of these languages, especially in terms of tractability and clarity, they nevertheless have one crucial shortcoming: with very few exceptions - see section 6 for a discussion of the only one we know which specifically addresses KBS design - they do not include 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, because 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 even in the case of a single designer.

4.1. OCML-DDL—A decision based design language for KBS design

To overcome this problem (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. 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. In contrast with formal approaches to KBS design, for instance $K_{BS}SF$ (Jonker and Spee, 1993), OCML-DDL provides a semi-formal means of depicting and manipulating the design model at the knowledge level.

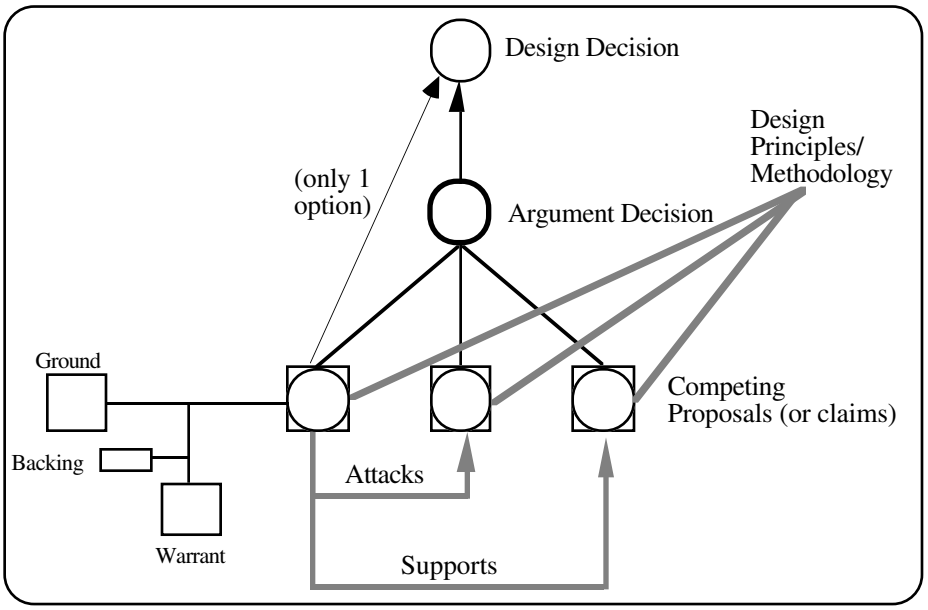

Figure 4: A model of design argumentation

The model for design argumentation (as presented in Figure 4) combines aspects of the IBIS and Toulmin models of argumentation with aspects of Gruber and Russell's theory (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 competing 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, for example, conflicting consequences of applying design principles (where there are applicable principles available). The principal features of this model are:

- It relates design decisions to decisions about competing arguments.
- The arguments are contextualised in that they are about particular *issues* or design decisions.
- The semantics of the design model is given (at least in part) by the arguments recorded in the design decisions.
- Unlike the model given by Gruber and Russell, there is no automatic means of generating the rationales.
- Unlike the model given by Ramesh and Dhar, changes in the argumentation are not automatically reflected in changes in the target model.

In the VITAL approach to the design of KBSs, the designer follows a series of design steps (see below) some of which involve choices between competing possibilities. Although choices are usually made on the basis of design principles (taking into account any operational design constraints), the designer may record reasons for design choices which do not refer to the principles already available. These may form the basis for new design principles.

As shown in Figure 5, proposals, choices 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 5 for explanatory reasons.

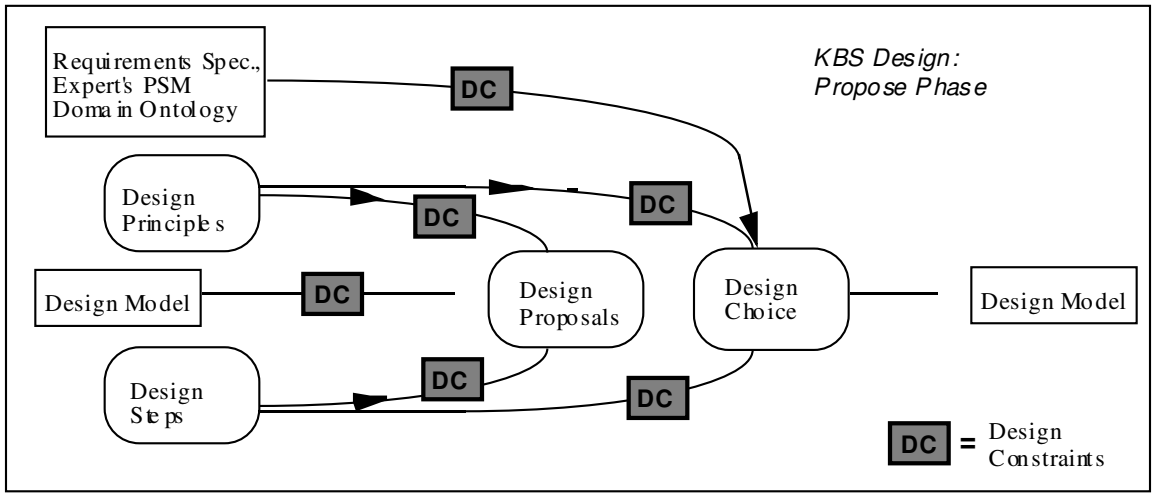

Figure 5: Design as argumentation

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 is 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. The design model thus includes design components (tasks, roles and so on) and a design rationale.

4.2. 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."

Our design principles fulfil most of the roles of Van de Velde's operators while also allowing them to be non-transformative guides to decision-making. They can for instance aid in choosing a new design focus as a result of which there are no changes in components of the design model. Design principles are 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, although they are not necessarily operational, they can often act as (meta-)rules for making a choice between competing design decisions or for transforming one design model into another. When they are 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).

- *Design Activity*. Design principles can act as: design steps (they generate new model expansions); justifications for design choices; 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 principles 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.

Design 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 invocation 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 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 1: Examples of design principles and their indices.

Table 1 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. Indeed this is often necessary because of the inconsistencies between principles.

The Design Principles act as ways of transforming the components of the target model (or at least shifting the design focus and so on) 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 by providing a standard reference model for recording design decisions (forming the basis for new design principles). 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 1 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.

4.3. 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. In particular, when there is no provision for model extension procedures and fixes in the specification of parametric design tasks we need to map the PSM's roles and domain requirements to the knowledge structures in the domain, as 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.

4.4. OCML-DDL—Mini-Manual for language

The following are the main elements in the OCML-DDL design decision language. We provide the OCML-DDL class definition and an informal description for each construct.

Design Decision

```
(def-class design-decision ()
      ((design-context :type design-context)
         (decided :type text)
         (justification :type text)
         (outcome :type model-transformation)
         (object :type model-component)
         (decided-by :type text)
         (date :type date)))
```

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. changes in the target model such as those in the particular object about which the decision is made or more general decisions about parts of the model or the model as a whole. The outcome may also be that the design focus shifts to a particular object or model part—a change in the rationale rather than the design components. The object slot refers to a model component, a partial model or to the model as a whole. Provision is also made for various annotations. Since decisions are made in particular contexts a design decision may record the current design context.

Design Context

```
(def-class design-context ()
      ((context-name :type name)
         (focus :type model-component)
         (goal :type text)
         (operative-constraints :type constraint-list)
         (agenda :type 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 which are operational (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), and an agenda which contains undischarged design tasks. As such the design context is dynamic and not part of the permanent design history.

Argument Decision

```
(def-class argument-decision ()
      ((design-context :type design-context)
         (accepted-proposal :type text)
         (competing-proposals :type list)
         (deciding-factor :type text)))
```

The argument decision structure records the accepted proposal and the deciding factor between it and other possible proposals. The deciding factor may refer to any design principles involved and will often figure as the justification in the corresponding design decision

Competing Proposal

```
(def-class competing-proposal ()
      ((design-context :type design-context)
         (proposal :type text)                          ;;claim
         (pro :type text)                               ;;grounds
         (contra :type text)                            ;;possible rebuttal
         (warranting-principle :type text)              ;;warrant
         (principles-involved :type name)               ;;backing
         (inference-type :type (deduction analogy))
         (supports :type design-proposal)
         (attacks :type design-proposal)
         (restates :type design-proposal)
         (participant :type text)))
```

The competing proposal is the structure which records the extended Toulmin argument. As well as the design context, the pro and con arguments, the warranting-principle (whether this is drawn from the pre-existing principles or newly created), the body of principles involved (those from the VITAL methodology may differ from those for another methodology) 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, and the recording of the participant, means that the Toulmin model is capable of recording complex multi-participant arguments as well as the details of each micro-argument.

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. (1995). 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). This scheme is discussed in detail in Zdrahal and Motta (1994). In what follows we present a series of design steps reconstructed from our solution. 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, 1990) all design methods can be seen as variations of the propose-evaluate-revise schema, whose I/O specification is shown below.

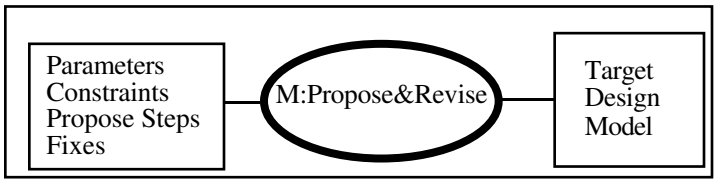

Figure 6: The I/O specification for the class of Propose&Revise PSMs

The relevant Design Decisions are straightforward:

```
(def-instance dd1 design-decision
      ((decided "Select initial PSM")
         (justification "Step 4a in VITAL design steps")
         (object Initial-PSM)
         (decided-by A.S.)
         (date 8-93)))

(def-instance dd2 design-decision
      ((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, 1994) 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 7.

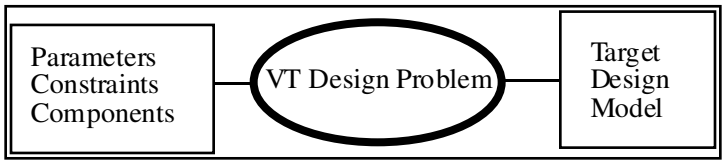

Figure 7: Functional view of the Ontolingua VT KB.

It is obvious that the mapping between the functional specification of the Propose&Revise class of methods shown in Figure 6 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.

```
(def-instance dd3 design-decision
      ((decided "Acquire knowledge about fixes")
         (justification "Needed by P&R PSM")
         (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:

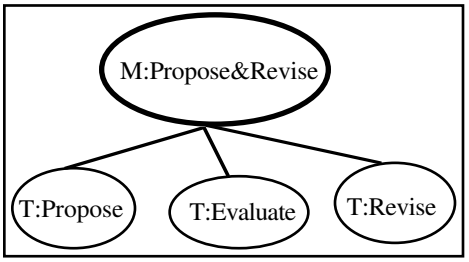

Figure 8: The task structure of the Propose&Revise PSM

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.

```
def-instance dd4 design-decision
      ((decided "Focus on the Propose sub-task")
         (justification "In general decomposition is depth-first left-right")
                        ;see Table 1
         (outcome focus(Propose))
         (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.

```
def-instance dc1 design-context
      ((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.

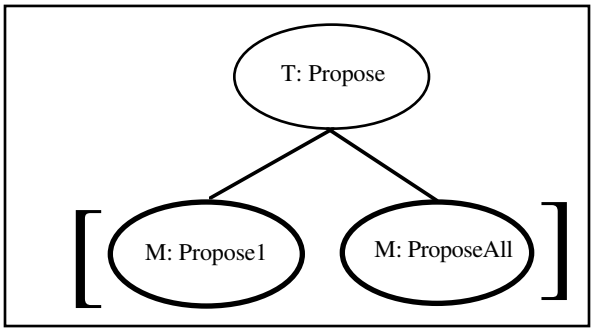

Figure 9: The possible methods for the Propose task

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 (1994) 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.

```
(def-instance dd5 design-decision
      ((decided "Use the ProposeAll method")))
(def-instance ad1 argument-decision
      ((accepted-proposal "Use the ProposeAll method")
         (competing-proposals "Use Propose1" "Use ProposeAll")
         (deciding-factor "Efficiency can over-ride requirements
            specification")))
(def-instance cc1 competing-proposal
         (proposal "Use Propose1")
         (pro  "Yost page 2")
         (warranting-principle "Follow specification")
         (principles-involved #vital-principles)))
(def-instance cc2 competing-proposal
         (proposal "Use Propose1")
         (pro  "ProposeAll implements the CMR strategy, which allows us to
        exploit the interdependencies between constraints and fixes")
         (warranting-principle "Be efficient")
         (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.

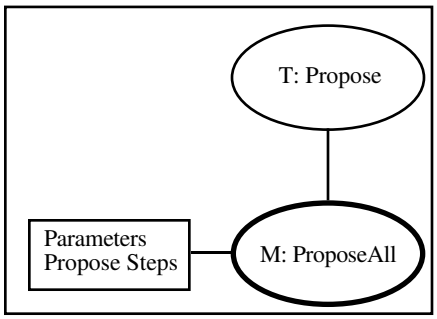

Figure 10: The ProposeAll method is chosen

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).

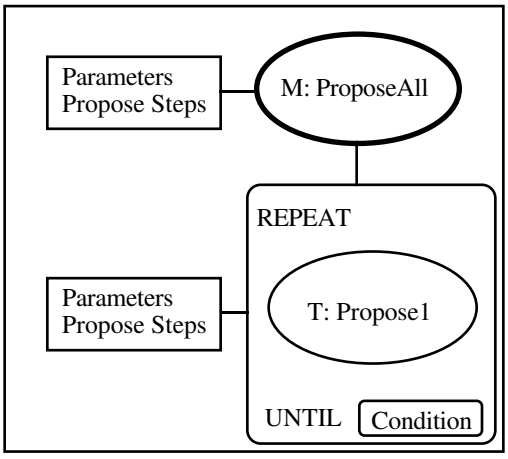

Figure 11: The task structure of ProposeAll

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).

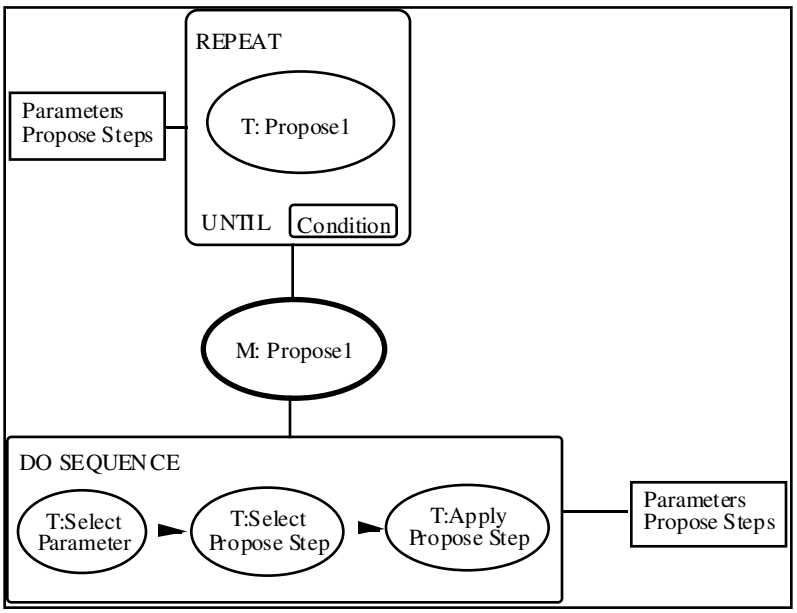

Figure 12: The task structure of Propose1

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. 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 13: 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.

This process is continued until the full problem-solving model (discussed in Motta et al., 1995) is reached.

6. RELATED WORK

DesignKARL OCML-DDL seeks to support the *process* of designing KBSs as well as the recording of design rationales. This process-based philosophy also underlies the DesignKARL formalism (Landes and Studer, 1994). DesignKARL uses a logic-based formalism for the representation of design components. OCML-DDL aims to provide an operational, upward-compatible subset of Ontolingua. DesignKARL includes extensions for the capture of design decisions (seen as relations "between two versions of parts of the system description" (ibid.: 33-15)) which can be used in prototyping the target system. Although it is not clear from the paper, it is likely that the rationale behind design decisions in DesignKARL will also be in a tightly constrained logic-related formalism (which may be computable in some manner). Since OCML-DDL constructs are intended to provide an easily accessible record of argumentation, they do not need to be as tightly constrained as those in DesignKARL. Design decisions in OCML-DDL (while usually related to design principles) can introduce other grounds in their rationales. Furthermore decisions and justifications can be expressed in natural language. This means that, while they are less amenable to computational manipulation, they are a more natural and expressive means of capturing actual design decision-making.

DRL The Decision Representation Language discussed by Lee and Lai (1991) has some features in common with OCML-DDL. It provides constructs for decision problems (design decisions), alternatives (competing arguments), claims (proposals) and relationships between claims (support and attacks links). Unlike OCML-DDl, DRL also includes a mechanism for breaking the decision problem into sub-problems (referred to as goals and sub-goals). This mechanism is not needed in OCML-DDL since goal decomposition is stored as part of the (operational) design principles (i.e. as a design strategy rather than an argumentational strategy). OCML-DDL is also more specialised for the design of KBSs (given its relation to design principles derived from knowledge engineering practice).

7. CONCLUSION—OCML-DDL CAN RECORD THE EXPERTISE OF A KE COMMUNITY

We believe that OCML-DDL offers a means for capturing design decisions which is natural, succinct and expressive. 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). This approach—basing the tool on an empirical study—is that adopted 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.

ACKNOWLEDGEMENTS

We would like to acknowledge the numerous discussions (sometimes argumentative) we have had with members of the Knowledge Media Institute on solutions to the Sisyphus-II problem.

REFERENCES

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

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

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

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

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

Fischer, G., Girgensohn, A., Nakakoji, K. and Redmiles, D. (1992). 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. (1991). Making Argumentation Serve Design. Human-Computer Interaction. 6. 393-419.

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

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

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

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

Gruber, T.R. and Russell, D.M. (1991). 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.

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

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

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

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

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. (1991). What's in Design rationale? Human-Computer Interaction, 6. 251-280.

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

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

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

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. 161–205.

Motta, E. (1995). KBS Modelling in OCML. Proceedings on the Workshop on Modelling Languages for KBS , Vrije Universiteit Amsterdam, January 30-31, 1995.

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

Motta, E., O'Hara, K., Shadbolt, N., Stutt, A. and Zdrahal, Z. (1995). Solving VT in VITAL: A Study in Model Construction and Knowledge Reuse. Submitted for publication.

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

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

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

Rathke, C. and Negele, A. (1991). 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. 147-159.

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

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

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

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

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

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

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

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

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

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

Yost, G. R. (1994). Configuring Elevator Systems . *DEC Project Report.* Available through anonymous ftp from ftp://camis.stanford.edu/pub/protege/s2-word.sea.hqx

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