A library of problem-solving components based on the integration of the search paradigm with task and method ontologies

Enrico Motta and Zdenek Zdrahal

Knowledge Media Institute
The Open University
Walton Hall, Milton Keynes,
MK7 6AA, United Kingdom
E.Motta, Z.Zdrahal@open.ac.uk

Abstract. In this paper we investigate the reuse of tasks and problem solving methods and we propose a model of how to organize a library of reusable components for knowledge based systems. In our approach we first describe a class of problems by means of a *task ontology*. Then we instantiate a generic model of problem solving as *search* in terms of the concepts in the task ontology, to derive a task-specific, but method-independent, *problem solving model*. Individual problem solving methods can then be (re-)constructed from the generic problem solving model through a process of *ontology/method specialization* and *configuration.* The resulting library of reusable components enjoys a clear theoretical basis and has been tested successfully on a number of applications. In the paper we illustrate the approach in the area of *parametric design*.

1. INTRODUCTION

A *problem solving method* (PSM) is a domain-independent specification of the reasoning process of a *knowledge-based system* (KBS). PSMs play an important role in both the analysis and design phases of the KBS development life-cycle. In particular, they can be used as i) model-based templates to guide the knowledge acquisition (KA) process (van Heijst et al., 1992) and ii) to develop robust and maintainable applications by reuse (Marcus, 1988; Runkel et al., 1996; Motta, 1998b).

While all PSMs are obviously designed with some class of problems in mind, it is common to find in the literature a distinction between two main categories of PSMs, which, somewhat ambiguously, are termed *task-specific* and *task-independent.* The former are PSMs designed to tackle a particular class of KBS problems - e.g. diagnosis or configuration design - and reflect such a commitment through the use of a problem-specific terminology. Examples of task-specific PSMs include the suite of diagnostic PSMs developed by Benjamins (1993) and the PSMs for design described by Chandrasekaran (1990). Throughout this paper we will use the term *problem type* (Breuker and van de Velde, 1994) to refer to the high-level generic tasks, e.g. parametric design or fault diagnosis, which are tackled by task-specific PSMs.

Task-independent PSMs do not subscribe to a problem-type-specific terminology but specify reasoning steps either in terms of a generic problem solving paradigm, such as *search* (Newell and Simon, 1976), or in terms of the *epistemological* properties of the domain knowledge base

(Beys et al., 1996)[1]. Hence, task-independent PSMs tackle problems characterized at a higher level of abstraction than task-specific ones. For instance, the A* search algorithm (Nilsson, 1980) can be viewed as a PSM for finding the cheapest path - according to some criterion - to a solution state in a state space.

Both approaches, as pursued in recent years, suffer from problems. Task-independent PSMs do not provide enough support for knowledge acquisition and, as discussed by Klinker et al. (1991), are more difficult to reuse than task-specific ones. On the other hand a task-specific terminology limits the possibility of reusing PSMs across classes of tasks and can impede our understanding of what a PSM really does. For example, in a number of papers (Zdrahal and Motta, 1995, 1996; Motta and Zdrahal, 1996; Motta, 1998b) we have analysed the *Propose&Revise* PSM developed by Marcus and McDermott (1989) and shown that a task-independent characterization of this PSM as a search algorithm makes it possible not only to clarify the computational basis for the knowledge structures employed by the PSM, but also to explain its incomplete nature (see also discussion in section 4.1.2).

Thus, we believe there is a need for an approach to PSM specification and library organization which can reconcile the advantages in terms of KA and reuse afforded by task-specific formulations with the clear theoretical foundations and problem generality provided by task-independent problem solving paradigms, such as search. Specifically, our approach relies on the following three key ideas:

- We use different kinds of *formal ontologies* (Gruber, 1993) to specify the generic structure of a class of problems (*task ontologies*) and the knowledge requirements of PSMs (*method ontologies*).
- All problem solving methods applicable to a class of problems, say P, are characterized as refinements of a common, task-specific, but method-generic *problem solving model*[2]. This model comprises a set of generic problem solving components, generic (sub-)tasks and (sub-)methods, which provide the high-level building blocks necessary to construct PSMs applicable to P. The model is associated with a *generic method ontology*, which expresses the minimal knowledge requirements which have to be satisfied by a PSM applicable to P.
- In order to bridge the gap between the method and task 'dimensions' and to provide a task-independent foundation to a task-specific library of PSMs, the construction of the generic problem solving model discussed in the previous bullet, say M, is driven by a task ontology and by the selection of a *generic problem solving paradigm*. In particular we make use of the search paradigm. The advantage of this choice is that it does not constrain the range of PSMs which can be modelled as specializations of the model M - i.e. all PSMs can be seen as performing search[3].

Thus, as shown in figure 1, PSM development in our approach can be characterized as a three-stage process:

1. Formalize the problem type by defining the appropriate task ontology.

---

1 Specifically, Beys et al. show that the behaviour of a diagnostic method such as Cover&Differentiate (Eshelman, 1988), which is normally described using a diagnosis-oriented terminology, can be reformulated as a graph-searching algorithm and its *domain assumptions* expressed as statements on the topological structure of the graph.

2 Here we use the term 'problem solving model', rather than 'problem solving method', to emphasize that, like generic algorithmic schemas in conventional software, this model may not be fully specified - e.g. it may abstract from specific control regimes.

3 This statement does not imply that a problem solver effectively has to search (i.e. that it has to examine possible alternative paths to a solution). It only emphasizes that all problem solving behaviour can be modelled as a search through a state space. Hence, in contrast with approaches such as Soar (Laird et al., 1987), we do not use search as the basis for a computational problem solving architecture, but simply as a 'methodological device', which allows us to move from a problem specification to a generic problem solving model, without introducing additional problem solving commitments.

2. Construct a generic problem solving model associated with a problem type by instantiating a generic model of search in terms of the task ontology defined at step 1.
3. Characterize individual PSMs (or PSM components) as specializations/configurations of the task-specific, search-based problem solving model developed at step 2.

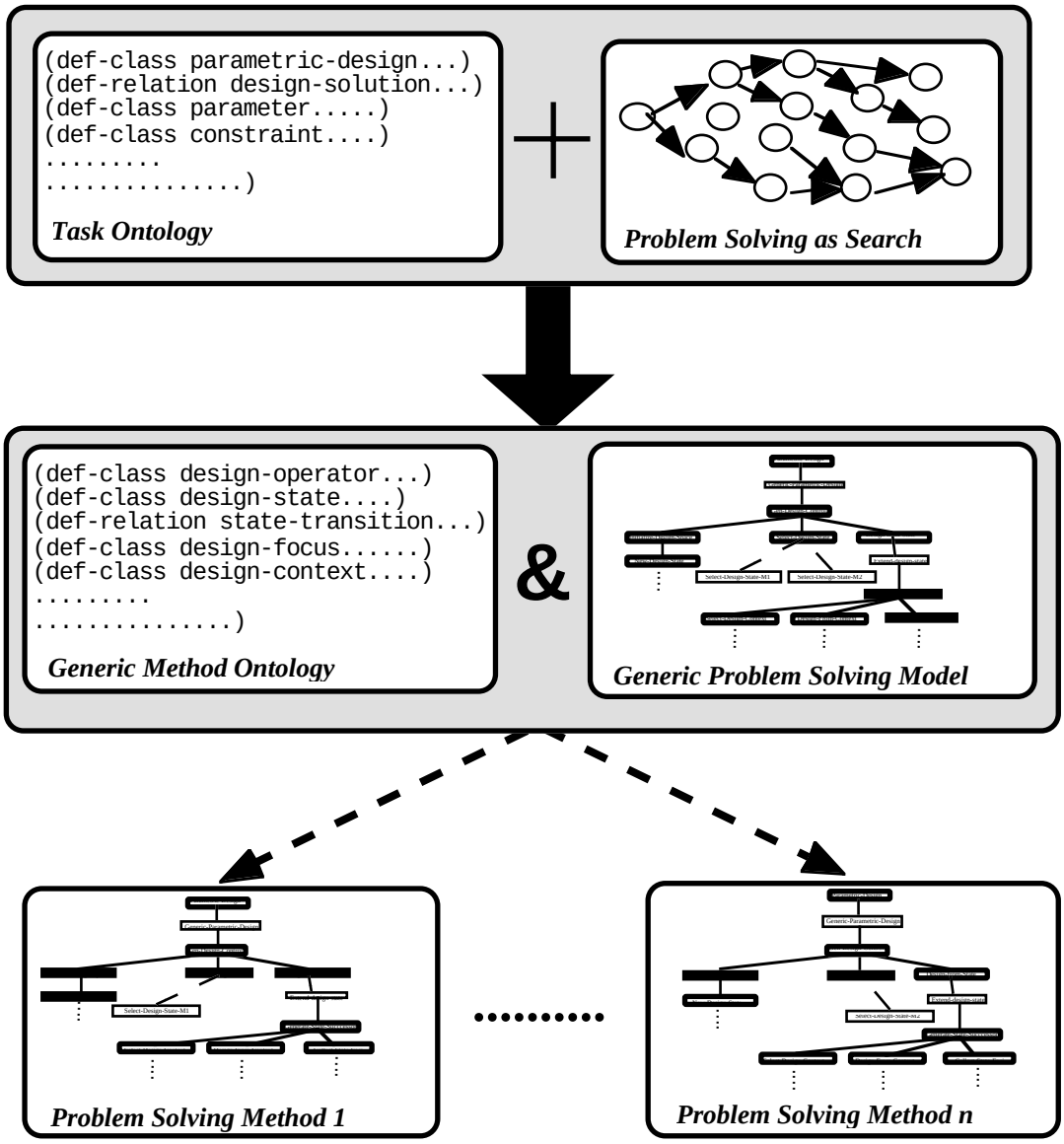

Figure 1. PSMs as refinements of generic problem solving model

In this paper we will illustrate these ideas in the context of a library of reusable components for *parametric design* problems (Wielinga et al., 1995; Motta and Zdrahal, 1996). The rest of the paper is organized as follows. In the next section we introduce the class of parametric design problems. In section 3 we present a generic model of parametric design problem solving, obtained by instantiating the search paradigm in terms of the parametric design task ontology. Then, in section 4, we will illustrate a library of PSMs for parametric design defined as specializations of the generic model presented in section 3. Section 5 discusses our approach to application development by reuse and briefly discusses some application models developed using our library of reusable components. Finally, in sections 6 and 7 we compare our work with alternative approaches to developing and organizing PSM libraries and we highlight some issues for future research.

2. A TASK ONTOLOGY FOR PARAMETRIC DESIGN PROBLEMS

2.1 Characterizing a class of problems

Problem solving activities concentrate on a limited number of entities in the world. The set of these objects, which can be either abstract or concrete, is called the *universe of discourse*.

In the universe of discourse the *problem* is characterized by at least two distinguishable states: the initial state and the final (goal) state. For example, the initial state of a medical diagnostic problem specifies all observable symptoms, a class of diseases and the patient's symptoms. The final state is characterized by an association of the symptoms with a particular disease. The key to defining generic and reusable problem specifications is that similar problems can be characterized in terms of a common, abstract universe of discourse, associated with a problem type.

A reusable conceptualization of a universe of discourse is called an *ontology* (Gruber, 1993). Here we use *task ontologies* to conceptualize the abstract universe of discourse associated with a problem type. In what follows we will illustrate a task ontology for parametric design problems.

2.2 Design as constructive problem solving

In general terms, design can be characterised as the process of producing a blueprint for an artefact from an input specification which includes a set of *requirements*, a set of *constraints* and some *building blocks*. Different classes of design problems can be identified by making different assumptions on the nature of the available building blocks. In particular, design problems where the set of available building blocks is fixed and known at the start of the design process are called *configuration design* problems (Stefik, 1995). As discussed by Mittal and Frayman (1989) the complexity of a configuration design problem can be greatly decreased by introducing assumptions about the structure of admissible solutions, such as the assumption that all solutions obey the same *functional template*. A stronger assumption postulates the existence of a *parametrized solution template*: this assumption reduces the general design problem to one of assigning values to design parameters in accordance with the given requirements, constraints, and *preferences*. This class of design problems is called *parametric design* (Wielinga et al., 1995; Motta and Zdrahal, 1996).[4]

2.3 Characterising parametric design tasks

A parametric design application can be defined as a mapping from a six-tuple <P, Vr, C, R, Pr, cf> to a set of solution design models, $\{D_{sol\text{-}1}, ..., D_{sol\text{-}n}\}$, where

P is a set of parameters, $\{p_1, ... , p_n\}$;

Vr is a set of value ranges, $\{V_1, ... V_n\}$, where $V_i = \{v_{i1}, ... , v_{il}\}$

C is a set of constraints, $\{c_1, ... , c_m\}$;

R is a set of requirements, $\{r_1, ... , r_k\}$;

Pr is a set of preferences, $\{pr_1, ... , pr_j\}$;

cf is a cost function;

---

[4] It is important to emphasize that our discussion here is only concerned with the *formal parametric design problem* - how to construct a solution design from a parametric design task specification. Naturally, there is more to design than deriving design models from input specifications. In particular, acquiring a task specification is itself a complex collaborative process, during which various stakeholders negotiate a common view of a design problem (Ehn, 1989; Greenbaum and Kyung, 1991). Moreover, this negotiation process, often called *problem framing* (Schoen, 1983), is typically an iterative process, which is intertwined with both problem solving and design evaluation (Bonnardel and Sumner, 1996). Hence, the fact that the work presented here is concerned exclusively with the formal design problem should not be taken as implying that the other aspects of the design process are less important, or that the design life-cycle can be characterized by means of a waterfall model, where design formulation and problem solving are carried out sequentially.

D_k is a design model, $\{<p_i, v_{ij}>\}$.

These concepts are discussed in the sections below.

2.3.1. Parameters and value ranges

A parameter specifies a primitive element of a design model. Each parameter, p_i, is associated with a value range, V_i, which specifies the set of legal values which can be assigned to p_i.

2.3.2. Constraints and requirements

Constraints specify conditions which must not be violated by a design. For instance, the VT elevator design application (Yost and Rothenfluh, 1996) includes constraints such as "The cab height must be between 84 and 240 inches, inclusive". Requirements specify properties which have to be satisfied by a solution - e.g. "the elevator should carry at least five people". In our formalization of parametric design problems we only enforce a conceptual distinction between requirements and constraints, rather than a logical one[5]. As pointed out by Wielinga et al. (1995), requirements have a 'positive' connotation, in the sense that they describe the desired properties of the target artefact, while constraints have a 'negative' connotation, in the sense that they limit the space of admissible designs, by expressing the applicable technological, physical, or legal restrictions. Moreover, constraints normally specify case-independent restrictions, while requirements tend to be case-specific. In general, we can say that both requirements and constraints express *design prescriptions*, which are essential to distinguish between *solution* and *non-solution* design models.

2.3.3. Preferences

Preferences describe task knowledge which ranks design models in accordance with some viewpoint. For instance, statements such as "Secretaries should be as close as possible to the head of group" and "Project synergy should be maximised", which are included in the specification of the Sisyphus-I office allocation problem (Linster, 1994), can be modelled as preferences. Formally, a preference is a relation of partial order defined on the space of design models. Different viewpoints correspond to different relations of partial order. Obviously, the partial order over design models determined by one preference may differ from that determined by another; in some cases different viewpoints may contradict each other.

2.3.4. Global cost function

A parametric design task specification typically includes a number of preferences which model various viewpoints for ranking solutions. In order to integrate the multiple preference criteria uncovered during the domain analysis we introduce the notion of *global preference*. A global preference, say gp, is a partial order relation which combines individual preferences in order to provide a uniform viewpoint for ranking design models[6]. For convenience, given that it is both common and useful to talk about the cost of a design model, we introduce a *global cost function* cf such that $cf(D_j) \leq cf(D_k)$ iff $<D_j, D_k> \in gp$. The value of the cost function is associated with the design model and therefore for any two design models the relation gp can be evaluated by comparing their associated costs.

2.3.5. Design models

A design model is a set of pairs $<p_i, v_{ij}>$ where p_i is a parameter and v_{ij} is the value assigned to p_i in the design model in question. We distinguish between different types of design models.

- A design model D_k is *complete* if each parameter in P has a value in D_k.
- A design model is *consistent* if it does not violate any constraint in C.

---

5 In general there is also a logical distinction between requirements and constraints: while requirements must be satisfied by a solution, constraints should simply not be violated (Wielinga et al., 1995). Because our characterization of parametric design problems assumes complete solution models, constraints are always applicable to solution models. Therefore the condition 'constraint not violated' is the same as 'constraint satisfied'.

6 Because individual preferences might not be mutually consistent, the process of acquiring a global preference is not simply concerned with the logical combination of multiple partial orders, but requires the various stakeholders in the the design process to negotiate a common criterion for ranking design models.

- A design model is *suitable* if it satisfies all requirements in R.
- A design model is *valid* if it is suitable and consistent.
- A design model is a *solution* if it is complete and valid.
- A design model $D_{sol\text{-}k}$ is an *optimal solution* if it is a solution and there is no other solution $D_{sol\text{-}j}$, such that $cf(D_{sol\text{-}j}) < cf(D_{sol\text{-}k})$.

In general, the goal of a parametric design problem is to find a solution design model. Of course, this goal can be further specialised for particular classes of parametric design applications, e.g. by requiring that an optimal solution be found.

2.4 The parametric design task ontology

In the previous section we have briefly described a specification of the class of parametric design problems. These ideas have been formally represented in a parametric design task ontology, which is part of our library of reusable modelling components. This ontology has been modelled in OCML (Motta, 1998a; 1998b), an operational modelling language, which provides constructs for specifying relations, functions, classes, instances, rules and control structures. Operationality is supported by means of a function interpreter, a control interpreter, and a proof system. The latter integrates inheritance and function evaluation with a backward chaining inference engine. OCML modelling is supported by a library of reusable definitions, which is structured according to the basic categories of our *application modelling framework*[7], i.e. *task*, *method*, *domain* and *application*. The library also relies on a number of *base ontologies*, which provide definitions for basic modelling concepts such as numbers, sets, relations, tasks, methods, roles, etc. Throughout the rest of this paper we will introduce OCML constructs 'opportunistically' when needed to explain relevant definitions. A complete description of the language can be found in (Motta, 1998b).

The parametric design task ontology comprises about 40 definitions and is fully described in (Motta, 1998b). Here we will illustrate the main modelling decisions taken when building the ontology.

2.4.1. Modelling the parametric design generic task

The OCML definition of the class of parametric design tasks is as follows:

---

7 This is discussed in section 5.1.

```
(def-class parametric-design (design-task) ?task
  ((has-input-role :value has-parameters
                   :value has-constraints
                   :value has-requirements
                   :value has-cost-function
                   :value has-cost-algebra
                   :value has-preferences)
   (has-output-role :value has-design-model :cardinality 1)
   (has-design-model :type design-model :max-cardinality 1)
   (has-parameters :type list :cardinality 1)
   (has-constraints :type list :max-cardinality 1)
   (has-requirements :type list :max-cardinality 1)
   (has-preferences :type list :max-cardinality 1)
   (has-cost-function :type cost-function :max-cardinality 1)
   (has-cost-algebra :default-value '(+ - <) :cardinality 1)

   (has-goal-expression
    :type legal-parametric-design-goal
    :default-value (kappa (?task ?design-model)
                     (design-model-solution ?design-model
                                            ?task)))))

(def-relation LEGAL-PARAMETRIC-DESIGN-GOAL(?rel)
  :iff-def (and (binary-relation ?rel)
                (subrelation-of ?rel
                                (inverse design-model-solution))))
```

A class of generic tasks is characterised in the OCML base ontology in terms of its *input roles*, *output role*, and *goal expression*. Roles specify the knowledge structures which are input or output to a task. A goal expression specifies a condition which can be used to determine whether or not a task has been carried out successfully. More precisely, the slot `has-goal-expression` in an OCML task indicates a binary relation which takes as first argument a task instance (e.g. a parametric design problem) and as second a possible solution to the task. Thus, the goal of a task, say ?t, is satisfied by a particular value, say ?v, if and only if the associated binary relation is satisfied by the pair <?t, ?v>. Roles are represented as task slots, as well as being explicitly listed either as values of slot `has-input-role` or `has-output-role`.

The definition of the parametric design class given above follows straightforwardly from the discussion in section 2.3: there are only two aspects which deserve attention. The first one concerns the inclusion among the input roles of a *cost algebra*. This is a triple which specifies the functions and relation to be used to merge and subtract design costs and to compare the costs of different design models. The second one concerns the specification of the goal of a parametric design problem. As pointed out earlier, the goal of a parametric design problem is to find a valid and complete design model. However, in practical design applications optimisation aspects are typically very important: given a set of requirements and constraints we normally wish to find the cheapest design which does the job. Therefore our definition specifies only a default goal for parametric design problems (to find a solution design model) and imposes a type constraint that a parametric design goal should be a sub-relation of relation `design-model-solution`[8]. In other words, this definition only imposes a minimal requirement on the characteristics of a solution design model. Subclasses or instances of this class are free to impose further restrictions on the space of feasible solutions.

---

8 The default goal of the task is represented by means of a *kappa expression*. This is a modelling construct allowing the specification of anonymous relations, much like *lambda expressions* support the specification of anonymous functions.

The relation `design-model-solution` models the relevant notion defined in section 2.3.5: a design model is a solution if and only if it is complete and valid. This relation is formally defined as follows.

```
(def-relation DESIGN-MODEL-SOLUTION (?dm ?task)
  :iff-def (and (design-model-complete ?dm
                  (role-value ?task has-parameters))
                (design-model-valid ?dm
                  (role-value ?task has-constraints)
                  (role-value ?task has-requirements))))

(def-relation DESIGN-MODEL-COMPLETE (?dm ?parameters)
   "A design model is complete if all the parameters are bound"
   :iff-def (not (exists ?x
                         (and (member ?x ?parameters)
                              (unbound-parameter ?x ?dm)))))

(def-relation DESIGN-MODEL-VALID (?dm ?constraints ?reqs)
  :iff-def (and (design-model-consistent  ?dm ?constraints)
                (design-model-suitable  ?dm ?reqs)))
```

Similarly, we can specify the other relations used in the above definitions.

2.4.2. Parameters and parameter values

We have seen that the goal of a parametric design task is to find an assignment of values to parameters which satisfies the given requirements and constraints. The association between parameters and values is typically modelled as a unary function (or binary relation) which associates a parameter directly to its value (Gruber et al., 1996). However, we have found that in practice a reasoner might handle several design models at the same time (i.e. several mappings between a parameter and a set of values) and therefore in our ontology this association is not direct but it is mediated by the chosen design model. The definition below formalizes this approach, by stating that a parameter ?p has value ?v in a design model ?dm if and only if the pair <?p, ?v> is an element of ?dm. The `:constraint` keyword is used to specify that ?v should be an element of the value range associated with ?p.

```
(def-relation HAS-VALUE (?p ?v ?dm)
  "Parameters have values w.r.t a particular design model"
  :iff-def (and (parameter ?p)
                (design-model ?dm)
                (element-of (?p . ?v) ?dm))
  :constraint (and (has-value-range ?p ?vr)
                   (element-of ?v ?vr)))
```

2.4.3. Modelling constraints and requirements.

As already pointed out, in our characterization of parametric design problems the difference between constraints and requirements is conceptual rather than formal. Therefore they are modelled in our ontology as subclasses of a generic `design-prescription` class. This is defined as follows:

```
(def-class DESIGN-PRESCRIPTION () ?c
  "The definitions common to constraints and requirements.
   A design prescription is characterised in terms of the associated
   expression.  This is a kappa expression predicated over a design
   model.  Not all design prescriptions are necessarily applicable
   to all design models."
  ((applicability-condition :default-value (kappa (?d) (true))
                            :type legal-prescriptive-expression)
   (has-expression :cardinality 1
                   :type legal-prescriptive-expression)))
```

There are two important aspects about this definition: design prescriptions are *reified* and are associated with an applicability condition. Reification is useful because it makes it easier to reason about constraints and requirements. The explicit specification of an applicability condition is necessary to ensure that requirements and constraints are checked only on the relevant design models - e.g. some requirements might only be applicable to complete models.

2.4.4. Modelling costs and preferences

Like design prescriptions, preferences are reified and associated with an expression. In the case of preferences this expression must be a backward clause associated with relation `prefer` - see definitions below. This solution makes it possible to define preferences in a modular way, while allowing maximum freedom for specifying arbitrarily complex preference expressions.

```
(def-class PREFERENCE () ?p
  "A preference defines an order over two design models.  The
   difference between a preference and a constraint or requirement
   is that these distinguish good from bad models, while preferences
   distinguish between better and worse models."
  ((has-expression :cardinality 1 :type prefer-expression)))

(def-class PREFER-EXPRESSION (proof-expression) ?exp
  "A prefer expression is a backward rule clause which tries to
   prove a prefer relation instance"
  ((proves-relation :value prefer))
  :constraint (and (== ?exp (?tail if . ?rest))
                   (== ?tail (prefer ?d1 ?d2))))

(def-relation PREFER (?d1 ?d2)
  "Use this relation to express preferences between design models"
  :constraint (and (design-model ?d1)(design-model ?d2))
  :axiom-def (defines-partial-order prefer))
```

A cost function is simply a function which associates a cost with a design model. Costs are typically represented either as real numbers or as vectors. The latter are useful to model cost functions which comprise a number of non-commensurable preferences. For instance, our model of the Sisyphus-I office allocation problem characterizes the output of the cost function as a four-dimensional vector, $<n_1, n_2, n_3, n_4>$, where n_1 measures the distance between the room of the head of the group and that of the secretaries; n_2 the distance between the manager's room and the rooms of the head of the group and the secretaries; n_3 the distance between the heads of projects and the head of group and the secretaries; and n_4 provides a measure of the 'project synergy' afforded by a solution. We use this notation to represent the relative importance of the various preferences elicited during the task specification process - e.g. that minimizing the distance between the head of the group and the secretaries is more important than the degree of project synergy afforded by a solution.

```
(def-class COST-FUNCTION (unary-function) ?cf
  "A cost criterion is a function which takes a design model and
   returns its cost.  The output can be either a real number or a
   vector"
  :iff-def (and (domain ?cf design-model)
                (range ?cf cost)))

(def-class COST () ?x
  "The costs we use are typically real numbers or vectors.
   This definition leaves other possibilities open"
  :sufficient (or (real-number ?x)
                  (vector ?x)))
```

3. A MODEL OF PARAMETRIC DESIGN PROBLEM SOLVING

As illustrated diagrammatically in figure 1, our approach uses the selection of a generic problem solving paradigm (search) as an epistemological device to bridge the gap between the task and method dimensions. In what follows we will illustrate how this approach can be used to develop a generic problem solving model for parametric design.

3.1 Parametric design as search

Searching in a parametric design context means to navigate a *design space* comprising a number of *design states*. These are uniquely defined by the associated design model.

In order to formulate a design space we need only a task specification, given that this provides all the necessary information for generating all possible design models associated with a task. No additional assumptions are introduced here, either about the structure of the design space or the availability of search-control knowledge. Thus, the notion of design space makes it possible to move from a task-oriented perspective to a problem solving-oriented one. Let us consider a design space about which we only know the generic structure of a node. It follows that, in the absence of additional knowledge, the only approach which a problem solving agent can take to solve the task is to *search*. Thus, by introducing the notion of design space, we introduce a problem solving framework which is completely method-independent and presupposes only the existence of a task specification. By instantiating this framework in terms of the concepts defined in the parametric design task ontology, we can then formulate a generic model of parametric design problem solving.

3.2 Design operators as primitive design steps

Each design state describes a certain model of the artefact. In the initial state the model is very vague since only design prescriptions are known. In a solution state the artefact is fully specified, all requirements are satisfied and no design constraint is violated. This *task-oriented* characterization of a design space describes the design space declaratively but does not provide any procedural instruments for defining transitions between states. Without means for state transitions a problem solving method would not be able to incrementally construct a solution to the task. Thus, the notion of *state transition* is crucial to introduce a problem solving element in our task-oriented definition of a design space.

Transitions between design states are achieved in our model by applying *design operators*. A design operator is a generic concept which represents an elementary design step. Specific design methods may specialize design operators by including additional problem solving knowledge. For instance, a Propose&Revise problem solver refines the concept of design operator by differentiating between those design operators used during the Propose task, *procedures*, and those used during the Revise task, *fixes*.

A design operator can be constructed in four possible ways:

- If the value range V_i of a parameter p_i is discrete, then a design operator for p_i can be defined as a generator which, given a design state in which p_i is unbound, produces alternative design extensions where p_i is bound to a different element of V_i.
- Functional constraints and requirements can be operationalized into design operators. For instance a functional constraint such as "door operator weight = door operator engine weight + door operator header weight" can be transformed into a design operator which calculates the value for parameter 'door operator weight'.
- If there is a preference, pr_i, which suggests a value, v_{ij}, for a parameter, p_i, then this preference can be transformed into a design operator extending the input design model with one which includes the assignment $<p_i . v_{ij}>$. When multiple operators exist for a particular parameter, then the relation `design-operator-order`, which is included in our generic method ontology, can be used to provide context dependent control knowledge. This mechanism makes it possible to transform preference ratings into search control knowledge.
- Heuristic problem solving knowledge can be brought in to construct an operator. For example, the recommended procedure for computing the position of the counterweight in the VT elevator design application places this half way between the platform and the U-bracket. In relation to the VT task specification, this operator does not define a constraint or a requirement. It could possibly be characterised as a preference, on the basis that locating the counterweight in a central position has some cost advantages. Another possibility is that the role of this operator is just to codify experiential problem solving knowledge - i.e. a central position is a 'good default' for the counterweight. In this case the knowledge expressed by the operator is not related to the task specification.

These four categories provide an epistemological characterization of design operators in terms of the type of knowledge they rely on. In addition, design operators can also be differentiated depending on whether they are used to extend a design model, to fix an inconsistency, or to lower the cost of a design model.

3.3 A generic problem solving model for parametric design

3.3.1. Basic goals and assumptions.

Our generic model of parametric design problem solving decomposes the parametric design task into a number of subtasks and proposes default (sub-)methods for carrying them out. Thus, its main purposes are (1) to identify the generic tasks which characterise parametric design problem solving, and (2) to provide the ‘root node’ of a library of PSMs for parametric design.

The first objective is based on the assumption that there exists a set of generic tasks, which is common to different methods for parametric design. This assumption is justified both by theoretical and empirical evidence. From a theoretical point of view the adoption of a search-centred framework constrains the number and the type of feasible problem solving activities. Empirical evidence is provided by existing surveys of design problem solvers (Balkany et al., 1993), which have uncovered generic problem solving activities which are common to different approaches.

The second objective has to do with providing a strong organizational structure for the library, which makes it possible to define new PSMs as relatively simple refinements/configurations of the generic problem solving model. In particular, PSM specification is carried out by performing any number of four types of activities: specializing one or more components in the generic problem solving model; refining the generic method ontology; selecting a particular sub-method for a sub-task; and introducing new sub-tasks or sub-methods. Because - as we will see shortly - i) the building blocks comprising the generic problem solving model are relatively high-level components and ii) each PSM is defined by refining a pre-existing model, typically only few components are needed to define new PSMs - on average six or seven. Moreover, the resulting PSMs are characterized in a homogeneous style - i.e. they share the same high-level components. This approach makes it easier both to compare and contrast PSMs and to define 'hybrid' PSMs by integrating components from pre-existing PSMs.

In section 4 we will discuss a number of PSMs, which were constructed by refining and augmenting the generic model for parametric design problem solving. In the remaining of this section we describe the main components included in the generic model.

3.3.2. Method-generic control regime.

Given the adopted search-based approach we can define a simple, but generically applicable control regime, which provides the main control structure of our problem solving model. This control regime - informally[9] specified in figure 2 - initializes the design space and then iteratively performs a cycle in which i) a design state is selected according to some criterion and ii) subtask Design-from-State is then invoked. This cycle is exited either when the design has been completed or when state selection fails.

```
Generic Task Generic-Design-Control
  Inputs:   Design-operators, Current-task
  Output:   Design-state
  Control:  Design-space
  Goal:     "To return a state which satisfies the goal of the current task"
  Subtasks: Initialize-Design-Space, Select-Design-State, Design-from-State
  Body:     Initialize-Design-Space(Current-task) -> Design-space
            Repeat
              Select-Design-State(Design-space) -> Design-state
              If "Select-Design-State fails"
                then Return () -> Fail
                else
                  If "Design-state satisfies the goal of the current task"
                    then Return () -> Success
                    else
                      Do
                        Design-from-State(Design-state)
```

Figure 2. Informal specification of Generic Design Control

The important feature of the control regime shown in figure 2 is that it is completely PSM-generic, i.e. all PSMs included in our library subscribe to it. Thus, it is possible to differentiate between alternative PSMs only on the basis of specific solutions to design subtasks, rather than in terms of the overall control regime. The advantage of this approach is that it is much easier to reason about functionally characterized behaviours than about different control regimes. Moreover, this approach also facilitates the process of mixing and matching problem solving components.

Of the three subtasks of Generic-Design-Control only Design-from-State and Select-Design-State introduce dimensions for characterising alternative PSMs for parametric design. Task Initialize-Design-Space simply initialises the design space by creating its root state. The creation of a new design state consists of two steps: (i) creating the association between the state and the relevant design model and (ii) evaluating this to derive the information which is needed to reason about this state and to compare and contrast it to other known states (e.g. during the state selection task). Tasks Evaluate-State, Select-State and Design-from-State are discussed in the following three sub-sections.

3.3.3. State evaluation.

There are four main types of knowledge associated with a design state which might be needed by a problem solver: *consistency* (whether the model violates some constraints), *cost*, *completeness* (whether any parameter is unbound in the model), and *feasibility* (whether the state can lead to a solution). This breakdown is meant to provide maximal coverage - i.e. given

---

9 We are giving an informal specification of this control task only for the sake of convenience. All model components in the library are specified in OCML - see (Motta, 1998b) for a full specification of the generic problem solving model for parametric design.

i) the characterization of the parametric design class of problems provided by our task ontology and ii) consistently with our experience, these four classes provide all the knowledge required to make decisions about the current design state. However not all problem solvers require all four classes of knowledge. For example, some problem solvers are not concerned with cost issues.

3.3.4. State selection.

At any stage of the design process, a problem solver (be it human or artificial) knows about a number of design states which are relevant to the problem in hand. Typically, these are the states which have been explored during the current design process, i.e. the states included in the portion of the design space searched so far. However, human designers are also capable of reusing past designs and the same applies to problem solvers which make use of case-based reasoning techniques when solving design applications (Zdrahal & Motta, 1996). Therefore, from a general point of view we assume that the input to task Select-Design-State refers generically to all the design states available to the problem solver, either because they have been explored during the current design process, or because the problem solver has access to other relevant design knowledge.

Assuming that a rational problem solver would not normally select a design state known to be unfeasible (a *dead end*), it follows that state selection is carried out in terms of the other three main criteria discussed in the previous section: completeness, consistency, and cost. In section 4 we will show that different state selection criteria account for the different behaviours of alternative PSMs - see also (Motta and Zdrahal, 1996).

3.3.5. Task Design-from-State

Having selected a design state (in practice, a design model), a problem solver has to decide how to modify it, which requires a decision on which operator to apply to the current model. In general, there are various levels of decision-making required in order to reach a decision on the selection of a design operator. At the highest level of abstraction operator selection is driven by the current *design context*. This can be seen as a high-level abstraction which is useful to denote typical design scenarios. For instance a Propose&Revise problem solver distinguishes between two kinds of operators: procedures and fixes. If the selected design model is inconsistent, then the problem solver may choose one of the relevant fixes, otherwise it will choose one of the applicable procedures. To account for this behaviour we say that operator selection in a Propose&Revise problem solver takes place in two different *contexts*: one in which design extensions are carried out, and one in which inconsistent models are revised.

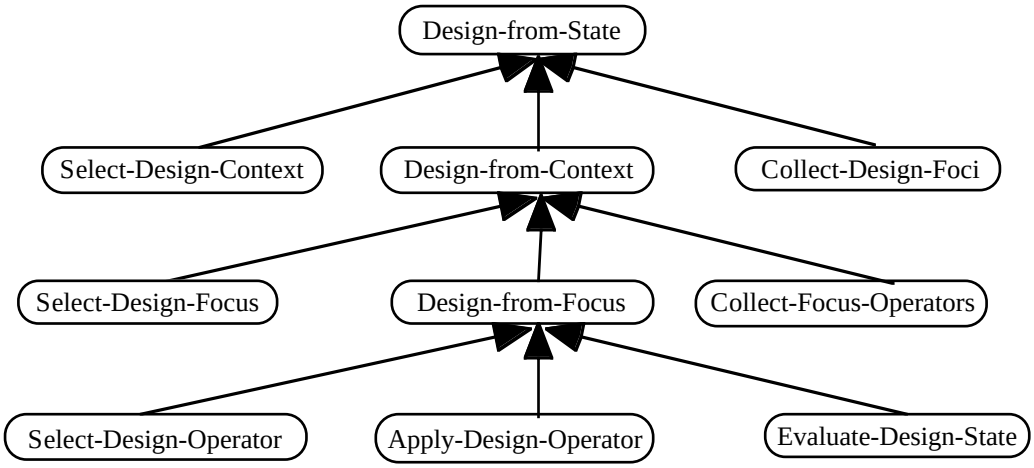

Figure 3. Main subtasks of Design-from-State

Having selected the current design context, a problem solver then decides on which element of the current design model to focus the design process. We call this the *design focus*. For example, in the context of the revision phase a Propose&Revise problem solver focuses on the particular constraint violation which it is trying to resolve. If the context is one of model

extension, then the design focus would be given by the parameter which is going to be assigned a value.

The notions of design context and design focus allow us to introduce two intermediate levels of decision making which are carried out after state selection and before the selection of a design operator. Thus, we can decompose task Design-from-State as shown in figure 3.

Space limitations do not allow us to discuss these subtasks in detail, therefore we only indicate the main decision-making aspects associated with them. Full details can be found in (Motta, 1998b).

The control method associated with task Design-from-State provides the main differentiation in terms of control between different PSMs. Methods such as Propose&Backtrack (Runkel et al., 1996) exhibit a simple control structure which repeatedly selects and extends the selected state. Instantiations of the Propose&Revise class of PSMs carry out different actions, depending on whether the current context is *propose* or *revise* and whether they intermingle the two phases or carry out revisions only after model completion (Zdrahal and Motta, 1995).

Tasks Design-from-Context and Design-from-Focus provide generic control regimes at different stages of the design process. The former selects a design focus and invokes task Design-from-Focus; the latter selects and applies a design operator and evaluates the resulting state.

Tasks Collect-Design-Foci and Collect-Focus-Operators are relatively uninteresting from a decision-making point of view: our model assumes that the relevant application-specific knowledge is provided. In most applications this knowledge consists of simple indexing mechanisms.

Much more interesting are the three selection tasks on the left hand side of figure 3. In particular, selecting the 'right' design focus, e.g. the right parameter or constraint violation, can be crucial for performance-related or competence-related reasons. From a performance point of view selecting the right focus helps to reduce unnecessary backtracking - in particular when carrying out design extensions. From a competence point of view, selecting the right focus is crucial when using incomplete PSMs. For instance, incomplete configurations of Propose&Revise problem solvers crucially rely on focus selection knowledge to reach a solution design; in (Motta et al., 1996) we showed that machine learning techniques can be used for acquiring focus selection knowledge for constraint selection in a revise context.

Our library provides both ontological and problem solving support for carrying out focus selection. In particular, it provides a default focus selection strategy (for a design extension context), which combines a *dynamic search rearrangement* heuristic (Dechter and Meiri, 1989)[10] with application-specific focus selection knowledge.

As already pointed out, our generic method ontology also provides support for expressing knowledge about operator selection. This type of knowledge is often called *local* preference knowledge (Poeck and Puppe, 1992) and is defined as knowledge which can be used to make locally optimal decisions. The use of local preference knowledge leads to greedy algorithms, such as hill-climbing, which can get stuck in local maxima. It is important to highlight that, in contrast with the focus selection case, the techniques developed in the constraint satisfaction literature, i.e. *value ordering heuristics* (Dechter and Pearl, 1988), are of only limited use here. These techniques try to find, at each stage of the variable assignment process, the least constraining values - i.e. the values which are less likely to cause backtracking at a later stage of the constraint solving process. Unfortunately these heuristics are only practical in binary constraint networks with fixed variable ordering. As discussed by Sadeh and Fox (1996), these heuristics do not perform very well in the presence of dynamic variable ordering, which is the general scenario assumed in our problem solving model.

---

10 Dynamic search rearangement "selects as the next variable to be instantiated a variable that has a minimal number of values which are consistent with the current partial solution. Heuristically, the choice of such variable minimizes the remaining choice" (Dechter and Meiri, 1989).

A listing of the main tasks included in the generic model of parametric design problem solving is given in table 1. The tasks are divided in four classes. *Goal specification tasks* are tasks which do not define a *task body* - i.e. they provide only a goal specification. Problem types are a subclass of goal specification tasks. *Composite tasks* introduce a task-subtask decomposition; *primitive tasks* solve a goal directly.

Tasks	**Type**
Parametric Design	Problem Type
Generic Design Control	Composite Task
Design from State	Goal Specification Task
Initialize Design Space	Composite Task
Select Design State	Goal Specification Task
Evaluate Design State	Composite Task
Evaluate feasibility	Primitive Task
Evaluate cost	Primitive Task
Evaluate consistency	Primitive Task
Evaluate completeness	Primitive Task
Extend design	Composite Task
Collect design foci	Goal Specification Task
Design from context	Composite Task
Select design focus	Goal Specification Task
Design from focus	Composite Task
Collect focus operators	Goal Specification Task
Order focus operators	Primitive Task
Select design operator	Goal Specification Task
Try design operator	Goal Specification Task
Apply design operator	Primitive Task

Table 1. Main tasks in parametric design model.

The tasks shown in table 1 provide a set of high-level building blocks for constructing parametric design problem solvers. In our experience this set is quite complete. In particular we found that the only situation in which a new task needed to be added was when introducing new contexts. For instance, in order to model Propose&Revise we needed to add a task, Revise-Design, associated with a revise context.

Table 2 shows a synoptic description of the main knowledge roles included in the generic method ontology associated with our model of parametric design problem solving. The classes shown in bold indicate the main domain roles associated with the framework. These roles can be filled by means of the appropriate application-specific knowledge, much as in the approach based on *role-limiting methods* (Marcus, 1988). Other types of domain knowledge, which are shown in bold-italics, denote optional roles, which are useful to improve the efficiency of the problem solving process, but are not essential to develop an application. Finally, the roles shown in plain text indicate intermediate knowledge structures generated during a problem solving process.

Knowledge Classes	Description
Design Operator	Knowledge for modifying design models
Design Space	The space of all design models considered by a problem solver
Design State	An element of the design space
Design Context	Abstract label associated with a design state which can be used to decide the next problem solving step.
Design Focus	Abstract notion which denotes the main design element driving the selection of a design operator.
Focus Selection Knowledge	Knowledge used to select a design focus
Operator Selection Knowledge	Knowledge used to select a design operator
Available Parameter Values	Knowledge which supports the generation of the values available for an unbound design parameter.

Table 2. Main classes of problem solving knowledge for parametric design.

4. ORGANIZING A LIBRARY OF PROBLEM SOLVING METHODS

The model of parametric design problem solving presented in the previous section is defined in terms of a number of generic tasks which provide useful building blocks for re-engineering existing problem solving methods for parametric design and for constructing new ones. In this section we will use these building blocks to model a number of parametric design methods. In particular, we will show that this uniform view of problem solving methods provides a number of advantages, including: (i) a common framework suitable for comparing and contrasting different methods, (ii) an organizational schema providing the overall structure of a library of reusable problem solving components and (iii) a search-centred interpretation model which can be used to understand the problem solving role played by the mechanisms and knowledge structures employed by problem solving methods - e.g. the fix mechanism in Propose&Revise.

4.1 Structuring problem solving methods

In what follows we will characterize a number of PSMs in terms of a generic method description template which considers a method's knowledge requirements, its approach to state selection and processing, and its global properties, such as completeness and optimality. We use this particular template for the simple reason that it provides all the information we need to understand what a PSM can provide (*global properties*), what knowledge it requires (*knowledge requirements*) and how it behaves (*state selection and processing*). Of course, it is important to keep in mind that the library does not comprise monolithic PSMs, but rather method components (e.g. generic tasks) and ontological definitions, which can be easily combined to produce a large number of alternative PSMs. Therefore, when talking about a "PSM in the library" we are really only indicating a particular configuration of method components which is interesting from some viewpoint.

Table 3 applies our method description template to a straightforward configuration of the problem solving model described in the previous section. This PSM, which is called *Generic-Parametric-Design*, was defined by augmenting the generic problem solving model with the primitive methods required for carrying out all goal specification tasks included in the model. For instance, these primitive methods include mechanisms for carrying out state, focus and operator selection. Generic-Parametric-Design makes use of a depth-first search strategy, as well as focus selection and operator selection knowledge. The former is used to minimize backtracking and the latter to perform local optimizations. This PSM only considers one design

context, *extend*, and its state selection policy always selects the maximal, consistent and feasible design model.

State selection & processing	
State Selection Policy	1) Violated constraints: No 2) Design model: Max
Contexts	Extend (Extend incomplete state)
Focus Types	Parameter
Knowledge requirements	
Design Operator Types	Design extension operator
Problem Solving Knowledge	Focus Selection Knowledge Operator Selection Knowledge Available Parameter Values (Cost Function not needed)
Global properties	
Reachable Design Space	All feasible states generated by the depth-first search.
Completeness, Optimality	Complete, optimizes design operator selection

Table 3. Synoptic description of Generic-Parametric-Design.

It is interesting to note that, because of the state-based control regime used, this PSM does not require a global cost function. At each cycle of the design process, the state selection mechanism will return at most one consistent and maximal state. Therefore any further cost-based discrimination would be unnecessary.

4.1.1. Propose&Backtrack.

Propose&Backtrack is the method used by Runkel et al. (1996), to solve the VT problem without resorting to the use of fixes. This method implements a simple depth-first control regime in which, at each step of the design process, unassigned parts are selected and assigned. The assignment is carried out by selecting a value from the value range of the selected parameter (part). If the assignment results in an inconsistency, a different value is tried. If there are no values left, chronological backtracking is used to go back to a consistent state. When deciding which value to assign to a part, Propose&Backtrack assumes the existence of local preference knowledge, which can be used to rank available parameter values. In the case of the VT application, this knowledge is based on the cost assigned to the relevant procedures and fixes.

As shown by the template in table 4, Propose&Backtrack is essentially the same PSM as Generic-Parametric-Design. Like the latter, Propose&Backtrack makes use of local preference knowledge to make 'good' design extensions, and on chronological backtracking to go back and explore alternatives paths to a solution, when an inconsistency or a dead end is encountered. Therefore, its performance relies on two crucial aspects of the problem: that the available local preference knowledge is effective in guiding the search process, and that the problem exhibits only a *weakly connected* (Sadeh and Fox, 1996) constraint network. These are pretty strong assumptions, which are rarely jointly satisfied except in relatively simple parametric design problems. For instance, chronological backtracking is too weak a control regime to tackle VT efficiently (Runkel et al., 1996), while we also found that the available local preference knowledge in the KMI office allocation problem[11] (Motta, 1998b) was

[11] This is a real-world office allocation problem which our institute faced when moving to a new building.

inadequate to achieve a good solution by means of Propose&Backtrack.[12] However, in those cases where the constraint network is only weakly connected, as it is the case with the Sisyphus-I office allocation problem (Linster, 1994), then Propose&Backtrack can be used to generate solutions to the problem quite efficiently, even if these are not necessarily optimal. In particular, in those cases where no application-specific focus selection knowledge is available, heuristic techniques, such as dynamic search rearrangement, can be effectively used to improve the performance of Propose&Backtrack.

State selection & processing	
State Selection Policy	1) Violated constraints: No 2) Design model: Max
Contexts	Extend
Focus Types	Parameter (Part)
Knowledge requirements	
Design Operator Types	Design extension operator
Problem Solving Knowledge	Preference knowledge for value ranges Available Parameter Values
Global Properties	
Reachable Design Space	All feasible states generated by the depth-first search.
Completeness, Optimality	Complete, optimizes design operator selection

Table 4. Propose&Backtrack

4.1.2. Propose&Revise.

A problem with the two methods described so far is that they both rely on a uniform problem solving approach. As Stefik (1995) points out, "Seldom does a single search method provide an adequate problem-solving framework for a complex task." In particular a uniform problem solving approach inevitably restricts the types of problem solving knowledge which can be applied to the problem. For this reason researchers have developed problem solving methods which distinguish between multiple phases and introduce a richer variety of knowledge structures. A famous example of such an approach is Propose&Revise (Marcus and McDermott, 1989), which differentiates between design extension and revision and introduces the appropriate knowledge roles for both phases.

Given that all design methods contain a model extension phase, the main contribution of the Propose&Revise class of methods is in the introduction of the Revise task, which modifies pre-existing assignments by means of special-purpose design modification operators called *fixes*.

Applying fixes can be understood as performing *knowledge-based backtracking* (Marcus et al., 1988). Another way to look at fix application is by adopting a *state-centred viewpoint.* If we take this view then the main novelty of a Propose&Revise approach is that it does away with the assumption that only consistent states can be found on a *solution path* (i.e. a path from an initial to a goal state). This principle of *constraint violation tolerance* opens up a number of possible strategies for design problem solving. For instance this principle can be instantiated in case-based design by relaxing the constraint that only consistent design models need to be stored in a library of cases. In such a scenario, a case-based design problem solver could select the design model in the library which most closely match the current specification (Zdrahal and

---

12 Specifically, this inadequacy was related to the non-monotonic nature of the cost function used for the KMI office allocation problem, which made it very difficult to assess the quality of a partial model. As a result, PSMs such as Propose&Backtrack, which rely on local preference knowledge, were not as effective as others, such as *Propose&Improve,* which perform improvement steps over complete models - see section 4.1.4 for a description of Propose&Improve.

Motta, 1996), regardless of consistency issues, and then repair eventual inconsistencies by means of *repair methods* (Minton et al., 1992).

As discussed in detail in (Zdrahal and Motta, 1995; Motta, 1998b), several approaches to design revision are possible within the basic Propose&Revise approach. However, we can abstract from specific architectures and characterize the class of Propose&Revise in generic terms. Such a characterization is shown in table 5.

State selection & processing	
State Selection Policy	1) Design model: Max 2) Violated constraints: Min 3) Cost: Min
Contexts	Extend, Revise
Focus Types	Parameter (Extend), Constraint (Revise)
Knowledge requirements	
Design Operator Types	Design extension operator, Fix (Revise)
Problem Solving Knowledge	Fixes, operator cost, available parameter values
Global Properties	
Reachable Design Space	All feasible states (Propose), Revision space (Revise)
Completeness, Optimality	Method-dependent

Table 5. Propose&Revise

As shown in the table, Propose&Revise refines our generic model by differentiating between extend and revise contexts and between design extension and design revision operators. This differentiation introduces flexibility in the problem solving process and allows for specialized problem solving knowledge. However, the most important aspect which emerges from the table concerns the state selection policy used by Propose&Revise problem solvers, which gives priority to the size of design models over cost and consistency. Intuitively, the idea of a Propose&Revise approach is that backtracking needs to be avoided: the currently most complete model should be operated on, even if it is inconsistent. Thus, we can say that Propose&Revise introduces a paradigm shift from a *consistency-oriented* to a *completeness-oriented* approach to design problem solving.

Issues of completeness and optimality cannot be discussed for Propose&Revise as a class but are associated with specific instantiations. For instance, the Propose&Revise method originally developed by Marcus and McDermott (1989) checks for inconsistencies after each model extension step and revises the design model as soon as an inconsistency is found. We call this control regime *Extend-Model-then-Revise (EMR)* (Zdrahal and Motta, 1995). As discussed in (Motta and Zdrahal, 1996) EMR prunes heavily the search space while not providing a sound converging criterion and is therefore an incomplete method. Motta (1998b) describes a modification of the EMR architecture, which gently degrades to depth-first search if a fix application fails (and is therefore complete).

Only local optimality is typically provided by Propose&Revise methods, such as EMR. However, it is possible to define an instantiation of the Propose&Revise approach which makes use of a global optimality criterion when searching the revision space. The resulting PSM, CMR-A*, is described in the next section.

*4.1.3. CMR-A**

An alternative to EMR is the *Complete-Model-then-Revise (CMR)* approach (Zdrahal and Motta, 1995), in which revision only takes place once the design model has been completed. An advantage of CMR is that because all constraint violations are tackled together, after the completion of the design extension process, it is therefore possible to reason about the relations

between constraints, parameters and fixes, and about the fix application process itself. For instance it is possible in a CMR approach to make use of techniques such as the *min-conflict* heuristic[13], which improve the efficiency of the constraint satisfaction process in the average case (Minton et al., 1992).

The CMR-A* method uses an A*-style search during the revision phase (Zdrahal and Motta, 1995). The important aspect of this PSM is that its state selection policy adopts a *cost-centred* strategy. Thus, the method sacrifices quick convergence criteria (choosing the maximal design state) for cost minimization. This method is characterized in table 6.

State selection & processing	
State Selection Policy	1) Design model: Max 2) Cost: Min
Contexts	Extend, Revise (Heuristic cost control)
Focus Types	Parameter (Extend), Constraint (Revise)
Knowledge requirements	
Design Operator Types	Design extension operator, fixes
Problem Solving Knowledge	Heuristic cost function, heuristic search control, fixes, operator cost, available parameter values
Global Properties	
Reachable Design Space	All feasible states generated so far
Completeness, Optimality	Complete, global optimization over the revise space

Table 6. CMR-A*

4.1.4. Propose&Improve

Another way of introducing differentiation in the problem solving process, without sacrificing a consistency-oriented approach is by means of the Propose&Improve class of methods (Motta, 1998b) - see table 7.

[13] This heuristic says: "when a number of constraint violations occur, then try to modify the variable that minimizes most conflicts" - i.e. kill as many birds as you can with one stone.

State selection & processing	
State Selection Policy	1) Violated constraints: No 2) Design model: Max 3) Cost: Min
Contexts	Extend, Improve
Focus Types	Parameter - Unassigned (Extend), Most expensive (Improve)
Knowledge requirements	
Design Operator Types	Design extension operator, design modification operator
Problem Solving Knowledge	Focus selection, operator selection, available parameter values, detailed cost function
Global Properties	
Reachable Design Space	All feasible states generated so far (Propose), Currently best state (Improve)
Completeness, Optimality	Complete, globally optimal with respect to 'improve' phase

Table 7. Propose&Improve

The basic idea underlying Propose&Improve is that optimality can be achieved or approximated by dividing the problem solving process into two phases: a 'propose' phase, which is concerned with finding a solution, and an 'improve' one which attempts to improve it. This problem solving method is an example of using a 'pick and mix' approach The propose phase is carried out as in a Propose&Backtrack method, i.e. as a search process driven by local preference knowledge, with backtracking when an inconsistent model is encountered. The 'improve' phase consists of a global hill-climbing process which identifies the solution components which are currently most expensive, and then uses specific improvement operators to modify them. As in the case of Propose&Revise, a Propose&Improve method can be defined as a specialization of Generic-Parametric-Design, simply by introducing the relevant contexts and operator types.

Propose&Improve is particularly suitable for parametric design problems in which optimality is an important solution criterion and which are characterized by a *dynamic cost function* - i.e. a cost function in which the cost of an assignment can only be fully evaluated once a number of other assignments have been completed. This situation often arises in resource assignment problems, such as timetabling and office allocation (Motta, 1998b).

4.2 Summing up

Earlier we said that the approach we have taken to constructing and organizing a library of reusable components for parametric design provides the following benefits: i) a common framework suitable for comparing and contrasting different methods, ii) an organizational schema providing the overall structure of a library of reusable problem solving components and iii) an interpretation model which can be used to understand the problem solving role played by the mechanisms and knowledge structures employed by different problem solving methods.

In the previous sections we have shown how different PSMs can be characterized as specializations of our framework and compared in terms of their knowledge requirements and their global and state-related properties. These descriptions can be used to understand and differentiate the behaviours of alternative methods. For example, as discussed in detail in (Motta and Zdrahal, 1996) it is easy to see that the better competence exhibited by CMR-A* over EMR and CMR in the VT application is due to the incomplete nature of the search policies used by EMR and CMR - in particular the non converging criterion used for state selection.

The uniform description of PSMs afforded by our framework also makes it possible to understand better the nature of the problem solving knowledge used by different PSMs. For instance, our framework characterizes fixes as specialized operators for a revision context and isolates this notion from specific approaches to searching the revision space. In particular, when we configured our generic model of Propose&Revise problem solving to develop a rational reconstruction of the EMR method used by Marcus et al. (1988) we found that the there was no need to introduce the distinctions between *incremental* and *non-incremental* fixes and between fixes and *fix combinations* discussed in (Yost and Rothenfluh, 1996). These distinctions do not denote types of problem solving knowledge which are relevant to a knowledge-level analysis of Propose&Revise; instead they specify a particular search strategy, which consists of navigating a revision space in a cost-conscious way (Motta, 1998b).

We have also shown that our framework provides engineering leverage supporting a specialization-oriented process of PSM specification. In particular, the notions of *design operator*, *design focus* and *design context* provide very powerful abstraction mechanisms for defining new PSMs. For instance, less than 10 definitions were needed to define an EMR-style PSM as a refinement of Generic-Parametric-Design. These definitions were introduced to define i) the class of fixes as a refinement of class `design-modification-operator`; ii) a method associated with task Design-from-State specifying the EMR control regime; and iii) the relevant methods required to specify focus and operator selection and collection in a revise context.

Figure 4 shows the space of the main classes of PSMs which we have modelled in our library. The taxonomy is based on two criteria: i) whether or not a method's state selection policy only considers consistent states and ii) whether a method pursues local or global optimization policies. Of course, the figure only reflects some interesting points in the space of the PSMs and is not meant to circumscribe the coverage of the library. Endless possibilities for PSM configuration are available, by 'mixing and matching' different method components (e.g. we have defined a Propose-Revise-Improve method). Moreover, the figure does not include a number of case-based PSMs which we have also developed (Zdrahal and Motta, 1996). To date we have built and made use of eighteen different PSMs out of library components.

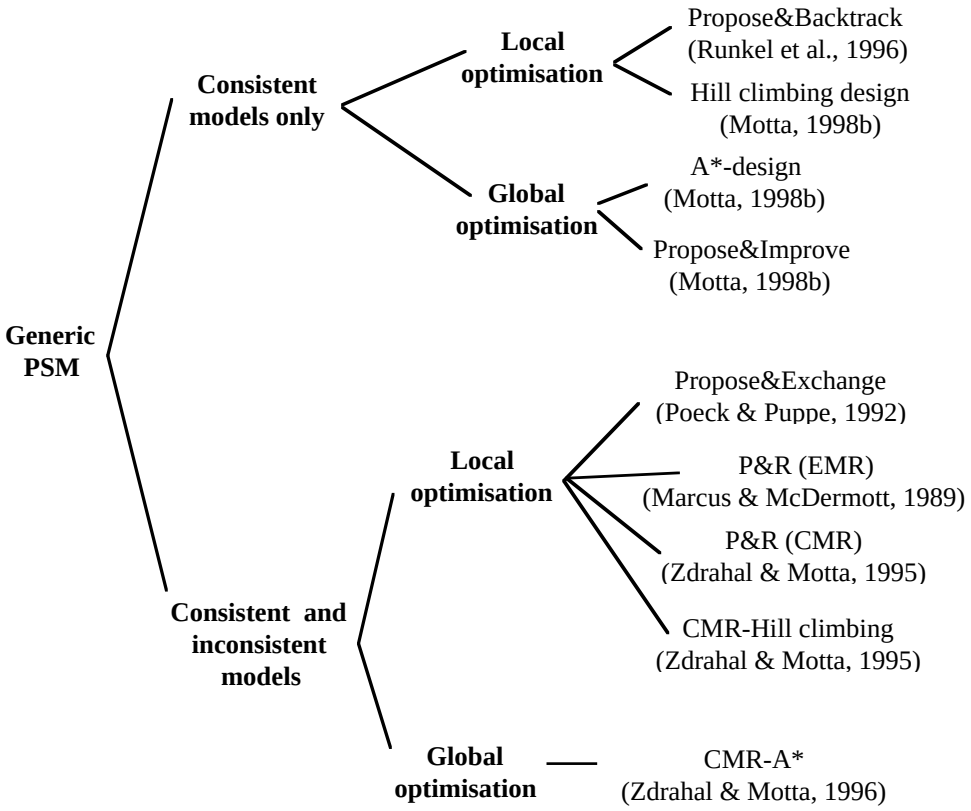

Figure 4. Taxonomy of problem solving methods for parametric design.

5. APPLICATION DEVELOPMENT BY REUSE

5.1 Integrating a problem solving method with domain knowledge

The modelling framework used for structuring the library construction process also supports a reuse-centred application development methodology. This consists of three major steps: i) instantiating a generic task ontology for a particular application, ii) selecting/configuring a suitable PSM and iii) integrating the chosen PSM with an application domain.

Domain knowledge itself can be specified in a task- or PSM-independent fashion, thus obtaining a reusable, *multi-functional* (Murray and Porter, 1988) knowledge base. Two problems can arise when applying a PSM to a multi-functional knowledge base:

- There can be a mismatch between the modelling schema used in the domain model and the knowledge types required by the problem solving method.
- The available multi-functional domain model might not comprise all the knowledge required by the problem solving method.

A representation mismatch can be addressed by adding appropriate *mapping mechanisms* (Gennari et al., 1994). For instance the method-specific concept of parameter can be mapped to the domain-specific concept of employee when applying a parametric design problem solver to an office allocation problem (Motta, 1998b).

The second bullet point concerns knowledge missing in the domain model. For example, in an office allocation problem the knowledge about office preferences and allocation requirements is important. This knowledge is application-specific and therefore cannot be part of a multi-functional domain knowledge base. Therefore, it needs to be acquired on an application-specific basis.

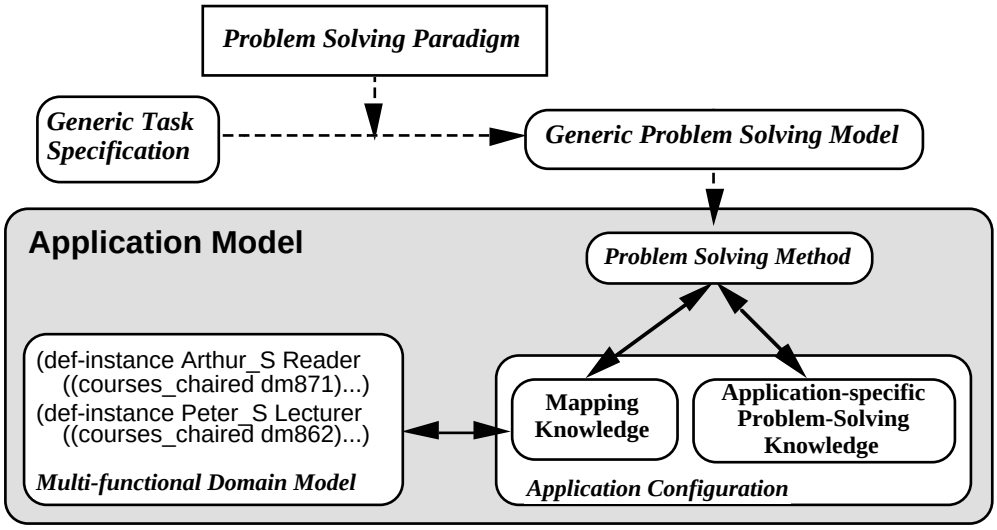

Figure 5. Overall modelling framework

In order to account for these two modelling situations, i.e. formulating the relevant mapping mechanisms and acquiring application-specific knowledge, our framework includes a fourth type of component, *application configuration knowledge*. In contrast with the other three components (task, problem solving method and domain model) this knowledge tends to be application specific and therefore not reusable. The overall application modelling framework is shown in figure 5.

5.2 Application domains

Our application domains include the Sisyphus-I (Linster, 1994) and KMI (Motta, 1998b) office allocation problems, the VT elevator design problem (Yost and Rothenfluh, 1996), sliding bearing design, initial vehicle design and design of casting technology for manufacturing mechanical parts (Valasek and Zdrahal, 1997).

Sisyphus-I is quite a simple problem which we used as a test case for trying out several PSMs (Motta, 1998b). In particular, we found that the dynamic search rearrangement heuristic was able to account for all but one of the problem solving steps of the virtual domain expert (Siggi). Moreover, by applying the *A*-design*[14] PSM to the problem, we found that the solution reached by Siggi was not optimal.

The reuse of various versions of Propose&Revise, originally developed for the VT problem, to the sliding bearing design problem was straightforward and required only simple domain mappings.

Tackling the KMI office allocation was more complex, given that the relevant optimality criterion (minimizing the distance between each KMI member and his collaborators across a number of 'affinity groups') was non-monotonic. In particular, we found that approaches based on local optimization did not produce very good solutions to the problem and that A*-type search was too expensive. As a result, we devised the Propose&Improve PSM, which performed much better than other alternatives.

Initial vehicle design and design casting technology for manufacturing mechanical parts are large applications being developed in cooperation with industrial partners. In all these cases application development includes selecting and adapting method components from the library both to make use of application-specific problem solving knowledge and to integrate additional tools (databases, simulation packages etc.). The preliminary results indicate that this technology leads to significant improvement in the efficiency of the design process (typical improvement by a factor 10).

---

[14] This is a PSM obtained by instantiating the A* search algorithm in terms of the parametric design task ontology.

6. RELATED WORK

6.1 Related work in modelling libraries

The most comprehensive generic library of model components is the one produced as a result of the Common KADS project (Breuker and Van de Velde, 1994). It consists of three main classes of library components: *modelling components*, *modelling operators*, and *generic models*. Generic models are "complete expertise models" (Valente et al., 1994); modelling components are elements of expertise models; and modelling operators are relations between generic models. These specify a possible transformation of a generic model. Modelling operators are included in the library to ensure that not only the results, but also the model building steps involved in a model construction exercise are captured by the library.

The generality of the approach taken in the Common KADS library essentially defines both the strengths and weaknesses of the Common KADS approach. It makes it possible to account for different approaches to modelling and to library organisation but necessarily it only provides fairly weak principles for structuring a library. In contrast with the CommonKADS approach, our library has a clear theoretical basis, which combines ontological engineering with a search model of problem solving.

A number of researchers have developed libraries of PSMs structured as task decomposition hierarchies (Steels, 1990; Chandrasekaran et al., 1992; Puerta et al., 1992; Van Heijst et al., 1992; Benjamins, 1993). While the details of the various approaches differ in various respects, the basic idea is essentially shared by all these approaches. Constructing a problem solving method for a specific application consists of recursively navigating a task-method decomposition tree, and at each stage selecting one of a number of possible methods applicable to a task. This selection can be done at run-time or during the design phase.

The basic principle of task-method structures is that, given a task, it is possible to find a number of methods which can be used to solve it. While this 'method-solves-task' association is adequate for the purpose of navigating a library and configuring a PSM, it does not, on its own, provide a strong enough organization model for developing a library. As a result it is quite difficult to get the task-method structure right (Orsvärn, 1996). In contrast with these approaches our library is based on a clear theoretical basis, which integrate both task-specific and task-independent foundations. Hence our library has a strong structure: any new method added to the library has to be formulated in terms of the overall model of parametric design problem solving. This approach is consistent with the principle of *method generality* suggested by Orsvärn: method adaptation is difficult and therefore should be avoided. Hence, PSMs should be as generic as possible. This principle, in its informal connotation, applies to the library of method components presented in this paper.

6.2 Related work in (parametric) design

6.2.1. Comparison with DIDS library

The DIDS system (Balkany et al., 1994; Runkel et al., 1996) provides domain-independent support for building design applications. The system is based on a generic model of configuration design problem solving, which defines the generic data structures and tasks, called *mechanisms* in the DIDS terminology, required for building design applications. The work presented here has a number of similarities with the DIDS approach. Both frameworks are based on a view of design as search through a design space and we share with the DIDS researchers the goal of generating a set of reusable components for design applications.

A difference between the model presented here and the DIDS library of mechanisms is that the latter aims to support full configuration design problem solving, while here we have focused on parametric design problems.

Another important difference is that we seem to subscribe to alternative views of what constitutes reuse. For the DIDS researchers reuse consists of providing a very general problem solving model. However, the price for such generality is inefficiency - see solution #1 to VT problem (Runkel et al., 1996). In contrast with the DIDS approach, we believe that supporting reuse consists of providing a rich set of reusable mechanisms, which can be used in different

problem solving scenarios to develop efficient problem solvers. This set is not meant to be minimal. On the contrary it is meant to be maximal and provide adequate leverage for developing efficient reasoners. For this reason our framework comprises a much more fine-grained breakdown of parametric design tasks than afforded by the DIDS tools and our library subsumes a variety of problem solving approaches, rather than just a generic constraint satisfaction paradigm.

Finally, a third difference between the DIDS approach and ours is that while our approach is based on a well-formulated model of problem solving, it is not obvious to see what is the principle/model underlying the library of mechanisms in DIDS. For example in (Balkany et al., 1993) the DIDS researchers analyse a number of configuration design systems, and try to classify the various mechanisms used by these systems into a number of generic categories: select design extension, make design extension, detect constraint violation, select fix mechanisms, make fix mechanisms, and test if-done. For each of these generic tasks a number of mechanisms drawn from the various systems are identified - e.g. 41 "make design extension" mechanisms are listed in the paper. The granularity of the mechanisms uncovered by this analysis varies significantly. Some mechanisms are low-level actions such as "do-step", which performs "one step of a task", others are fairly high-level ones such as "create-goal-for-constraint". The problem with such a bottom-up approach is that each system uses a different terminology, solves a different class of tasks - for instance arrangement vs. parametric design - and employs control mechanisms at different levels of abstraction. Therefore it is difficult to compare them. The approach we have taken here is different. We have specified exactly the class of tasks we are dealing with - parametric design - and then we have formulated the generic structure of the problem solving model appropriate for this class of tasks. This means that specific parametric design problem solvers can then be described by analysing how they carry out the generic tasks introduced by our framework.

6.2.2. Comparison with Wielinga et al.

Parametric design is formally analysed in (Wielinga et al., 1995). In this paper the authors define a parametric design task specification and refine the associated *competence theory* to derive properties of a CMR problem solving method. Such an approach succeeds in highlighting a number of assumptions underlying CMR, but it only produces limited results with respect to assessing its competence. The reason for these limitations is that, as already pointed out, the competence of CMR is directly related to its state selection strategy. Hence, it is not possible to characterise it without a framework which explicitly operates with these concepts.

Another important difference between the analysis of parametric design carried out by Wielinga et al. and ours concerns the approach taken to move from a task specification to a generic problem solving model. In their paper, Wielinga et al. use the Generate & Test problem solving paradigm to refine the task specification into an initial problem solving model. In this paper, we have instead adopted a view of design as search in a problem space. While search and Generate & Test have typically been considered as equivalent paradigms, it is interesting to note that a procedural approach leads to a quite different analysis of parametric design problem solving.

6.2.3. Comparison with constraint satisfaction approaches

Design problems in general and parametric design problems in particular can be viewed as constraint satisfaction problems and can be solved by means of constraint satisfaction techniques (Flemming et al., 1992). Therefore it is important to compare the framework presented here to constraint satisfaction models and discuss the differences between the two approaches.

In a nutshell the difference between the approach formulated here and constraint satisfaction techniques is that the former subscribes to a knowledge-based view of problem solving, where knowledge is brought in to tackle complexity (Lenat and Feigenbaum, 1987). Thus, the main goal of our framework is to identify the knowledge-intensive tasks and types of application-specific knowledge which can be exploited during parametric design problem solving to arrive quickly at a solution. In contrast with this approach, researchers in constraint satisfaction attempt to develop domain-independent techniques which can be applied to solve problems

characterised as assignments of values to variables which satisfy a set of constraints. Thus, there is a fundamental distinction in the goals driving research in knowledge modelling and research in constraint satisfaction and - of course! - like other researchers who subscribe to the knowledge-centred paradigm, we also believe that "the combinatorics of complex problems can best be handled through the use of domain-specific knowledge" (Wielinga and Schreiber, 1997).

Nevertheless, as we have pointed out when discussing the model, it is possible to integrate the techniques from the constraint satisfaction literature within a knowledge-intensive framework.

7. CONCLUSIONS AND FUTURE WORK

In this paper we have discussed a framework for application development and for organizing a library of reusable components. This framework draws from various KBS technologies/approaches, including ontologies, problem solving methods, search, knowledge acquisition as modelling and KBS reuse. In particular our approach identifies the different types of knowledge which comprise an application model, provides a clear theoretical basis to the development of a library of reusable components, and imposes a uniform model of problem solving which makes it easier to understand, compare, contrast and 'plug & play' problem solving components.

Our library now includes hundreds of definitions and it has been successfully used to build several applications. Its usefulness has been proved not only by constructing several applications by reuse, but also by showing that the PSMs themselves could be constructed as refinements of the generic problem solving model. No PSM in the library required more than a dozen additional components for its specification; typically, the number is between six and eight.

While all components of the library are specified in OCML, for the sake of efficiency we have also implemented a parametric design shell in LISP, which provides implementation-level components for all tasks and methods included in the generic problem solving model. Thus, efficient prototyping of application models, which integrate both OCML and LISP definitions, is also supported[15].

In the future we plan to extend this approach by tackling other problem types, such as diagnosis. As discussed in this paper, the search model of problem solving is completely generic and therefore we do not envisage any problem in applying this approach to other areas. Of course, in order to model diagnostic problem solving we expect that we will need to introduce a higher degree of differentiation in the state space (i.e. multiple types of contexts and operators) than it was needed for tackling parametric design.

Recent work by Fensel and Motta (1998) builds on the approach described here by characterizing PSM specification as a process of navigating a three-dimensional space consisting of *problem-solving paradigms*, e.g. search; *problem spaces*, i.e. task ontologies; and *domain assumptions*, i.e. method ontologies. This navigation process can be carried out by formulating the relevant *adapters* (Fensel, 1997), which formalize individual PSM-building steps. This work aims to provide a comprehensive theory of problem solving methods, subsuming both task-independent and task-specific approaches, and integrating knowledge-based development with 'conventional' software engineering approaches.

Finally, the library presented here provides the baseline resource for a large, collaborative research project, IBROW[3] (Benjamins et al., 1998), which aims to develop web-based tools supporting KBS construction by reuse.

ACKNOWLEDGEMENTS

This paper has benefited from in-depth criticisms/suggestions by Dieter Fensel, Bob Wielinga, Nigel Shadbolt, Tamara Sumner, John Domingue and several anonymous reviewers for the

---

15 In other words, the library is for real!

1998 Knowledge Acquisition Workshop and the International Journal of Human-Computer Studies.

REFERENCES

Balkany A., Birmingham W.P. and Tommelein I.D (1993). An Analysis of Several Configuration Design Systems. *AI in Engineering, Design, and Manufacturing*, 7, pp. 1-17.

Balkany , A., Birmingham W.P. and Runkel, J.T. (1994). Solving Sisyphus by Design. *International Journal of Human-Computer* Studies 40, pp. 221-241

Benjamins, R. (1993). *Problem Solving Methods for Diagnosis*. PhD Thesis, Department of Social Science Informatics, University of Amsterdam.

Benjamins, R., Plaza, E., Motta, E., Fensel, D., Studer, R., Wielinga, B., Schreiber, G. and Zdrahal, Z. (1998): An Intelligent Brokering Service for Knowledge-Component Reuse on the World-Wide-Web. In B. R. Gaines and M. Musen (editors), *Proceedings of the 11th Banff Knowledge Acquisition for Knowledge-Based System Workshop (KAW´98)*, Banff, Canada.

Beys, P., Benjamins, R., and Van Heijst, G. (1996). Remedying the Reusability-Usability Trade-off for Problem-Solving Methods. In B. R. Gaines and M. Musen (editors), *Proceedings of the 10th Banff Knowledge Acquisition for Knowledge-Based System Workshop (KAW´96)*, Banff, Canada, 9-14 November, 1996.

Bonnardel, N. and Sumner, T. (1996). Supporting Evaluation in Design. *Acta Psychologica, 91*, pp. 221-244.

Breuker, J. A. and Van De Velde, W. (1994). *CommonKADS Library for Expertise Modelling*. IOS Press, Amsterdam, The Netherlands.

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

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

Dechter, R., and Meiri, I. (1989). Experimental evaluation of pre-processing techniques in constraint satisfaction problems. *Proceedings of the 11th International Joint Conference on Artificial Intelligence - IJCAI '89*, pp. 271-277. San Mateo, CA, Morgan-Kaufmann.

Dechter, R., and Pearl, J. (1988). Network-based Heuristics for Constraint Satisfaction Problems. *Artificial Intelligence Journal, 34*, pp. 1-38.

Ehn, P. (1989). *Work-Oriented Design of Computer Artefacts*. Arbetslivscentrum, Stockholm.

Eshelman, L. (1988). MOLE: A Knowledge Acquisition Tool for Cover-and-Differentiate Systems. In S. Marcus (Editor), *Automating Knowledge Acquisition for Expert Systems*, pp. 37-80. Kluwer Academic Publishers.

Fensel, D. (1997). The Tower-of-Adapter Method for Developing and Reusing Problem-Solving Methods. In R. Benjamins and E. Plaza (Editors). *Knowledge Acquisition, Modeling, and Management. Proceedings of the 10th European Workshop, EKAW '97.* Lecture Notes in Artificial Intelligence 1319, pp. 97-112, Springer-Verlag.

Fensel, D. and Motta, E. (1998). Structured Development of Problem Solving Methods. In *Proceedings of the 11th Banff Knowledge Acquisition for Knowledge-Based System Workshop (KAW´98)*, Banff, Canada.

Flemming, U., Baykan, C. A., Coyne, R. F., and Fox, M. S. (1992). Hierarchical generate and test vs constraint-directed search. In J. S. Gero (Editor), *Artificial Intelligence in Design '92*, pp. 817-838. Kluwer Academic.

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

Greenbaum, J. and Kyung, M. (1991). *Design at Work: Cooperative Design of Computer Systems.* Lawrence Erlbaum Associates, Hillsdale, NJ.

Gruber, T. R. (1993). A Translation Approach to Portable Ontology Specifications. *Knowledge Acquisition, 5(2)*, pp. 199-220.

Gruber, T. R., Olsen, G. R., and Runkel, J. (1996). The configuration design ontologies and the VT elevator domain theory. *International Journal of Human-Computer Studies, 44(3/4)*, pp. 569-598.

Klinker, G., Bhola, C., Dallemagne, G., Marques, D., and McDermott, J. (1991). Usable and Reusable Programming Constructs. *Knowledge Acquisition 3*, pp. 117-136.

Laird, J., Newell, A., and Rosenbloom, P. (1987). Soar: An Architecture for General Intelligence. *Artificial Intelligence 33*, pp. 1-64.

Lenat, D. B. and Feigenbaum, E. A. (1987). On the Thresholds of Knowledge. *Proceedings of the Tenth International Joint Conference on Artificial Intelligence (IJCAI '87)*, pp. 1173-1182. Morgan Kaufmann, Los Altos, CA.

Linster M (1994). Problem statement for Sisyphus: models of problem solving. *International Journal of Human-Computer Studies 40*, pp. 187-192.

Marcus S. (editor) (1988). *Automatic Knowledge Acquisition for Expert Systems.* Kluwer Academic.

Marcus, S. and McDermott, J. (1989). SALT: A Knowledge Acquisition Language for Propose and Revise Systems. *Journal of Artificial Intelligence, 39(1)*, pp. 1-37.

Marcus, S., Stout, J., and Mc Dermott, J. (1988). VT: An Expert Elevator Designer that uses Knowledge-Based Backtracking. *AI Magazine 9(1), Spring Issue*, pp. 95-112..

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

Mittal, S. and Frayman, F. (1989). Towards a Generic Model of Configuration Tasks. *Proceedings of the 11th International Joint Conference on Artificial Intelligence - IJCAI '89*, pp. 1395-1401. San Mateo, CA, Morgan-Kaufmann.

Motta, E. (1998a). An Overview of the OCML Modelling Language. Paper presented at the 8th Workshop on Knowledge Engineering Methods and Languages (KEML '98). Available from URL http://kmi.open.ac.uk/~enrico/papers/keml98.ps

Motta E. (1998b) Reusable Components for Knowledge Models. *PhD Thesis*. Knowledge Media Institute. The Open University. UK. Available from URL http://kmi.open.ac.uk/~enrico/thesis/thesis.html

Motta E. and Zdrahal Z. (1996). Parametric design problem solving. In B. R. Gaines and M. Musen (editors), *Proceedings of the 10th Banff Knowledge Acquisition for Knowledge-Based System Workshop (KAW´96)*, Banff, Canada, 9-14 November, 1996.

Motta E., Stutt A., Zdrahal Z., O'Hara K. and Shadbolt N. (1996). Solving VT in VITAL: a Study in Model Construction and Knowledge Reuse. *International Journal of Human-Computer Studies 44(3/4)*, pp. 333-371.

Murray, K. and Porter, B. (1988). Developing a Tool for Knowledge Integration: Initial Results. *Proceedings of the 3rd Banff Knowledge Acquisition Workshop*, Banff, Canada.

Newell, A. and Simon, H. A. (1976). Computer Science as Empirical Enquiry: Symbols and Search. *Communications of the ACM, 19(3)*, pp. 113-126, March 1976.

Nilsson, N. J. (1980). *Principles of Artificial Intelligence*. Morgan Kaufmann, Los Altos, CA.

Orsvärn, K. (1996). Principles for Libraries of Task Decomposition Methods - Conclusions from a Case-study. In Proceedings of the European Knowledge Acquisition Workshop, EKAW'96. *Lecture Notes in Artificial Intelligence, 1076*. Springer-Verlag, pp. 48-65.

Poeck, K. and Puppe, F. (1992). COKE: Efficient Solving of Complex Assignment Problems with the Propose-and-Exchange Method. *5th International Conference on Tools with Artificial Intelligence*. Arlington, Virginia.

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)*, pp. 171-196.

Runkel, J. T., Birmingham, W. B. and Balkany, A. (1996). Solving VT by Reuse. *International Journal of Human-Computer Studies 44 (3/4)*, pp. 403-433.

Sadeh, N. and Fox, M. S. (1996). Variable and value ordering heuristics for the job shop scheduling constraint satisfaction problem. *Artificial Intelligence, 86(1)*, pp. 1-41, September 1996.

Schoen, D. A. (1983). *The Reflective Practitioner: How Professionals Think in Action*. New York, Basic Books.

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

Stefik, M. (1995). Introduction to Knowledge Systems. *Morgan Kaufmann.* 1995.

Valasek M. and Zdrahal Z. (1997). Experiments with Applying Knowledge Based Techniques to Parametric Design. In A.Riitahuhta (editor), *Proceedings of the International Conference on Engineering Design - ICED 97*. Volume 1, pp. 277-280, Tampere University of Technology, Finland.

Valente, A., Breuker, J. A. and Van De Velde, W. (1994). The CommonKADS Expertise Modelling Library. In Breuker, J. A. and Van de Velde, W., *The CommonKADS Library for Expertise Modelling*, pp. 31-56. IOS Press, Amsterdam, The Netherlands.

Van Heijst G., Terpstra P., Wielinga B. and Shadbolt N. (1992). Using Generalized Directive Models in Knowledge Acquisition. In Th. Wetter, K.-D. Althoff, J. Boose, B.R. Gaines, M. Linster and F. Schmalhofer (eds.) *Current Developments in Knowledge Acquisition - EKAW '92* (Springer-Verlag) pp. 112-32.

Wielinga B.J., Akkermans J.M. and Schreiber A.TH. (1995). A Formal Analysis of Parametric Design Problem Solving. In *Proceedings of the 9th Banff Knowledge Acquisition for Knowledge-Based Systems Workshop* (B. R. Gaines and M. Musen eds.). pp. 37-1 - 37- 15.

Wielinga B.J. and Schreiber, A.,Th. (1997). Configuration-design problem solving. *IEEE Expert, 12(2)*, pp. 49-56, 1997.

Yost G.R. and Rothenfluh T.R. (1996). Configuring elevator systems. *International Journal of Human-Computer Studies 44*, pp. 521-568.

Zdrahal Z. and Motta E. (1995). An In-Depth Analysis of Propose & Revise Problem Solving Methods. In B. R. Gaines and M. Musen (editors), *Proceedings of the 9th Banff Knowledge Acquisition for Knowledge-Based Systems Workshop*. Banff, Canada.

Zdrahal Z. and Motta E. (1996). Improving Competence by Integrating Case-Based Reasoning and Heuristic Search. In B. R. Gaines and M. Musen (editors), *Proceedings of the 10th Banff Knowledge Acquisition for Knowledge-Based System Workshop (KAW´96)*, Banff, Canada, 9-14 November, 1996.