A Formal Analysis of Parametric Design Problem Solving[*]

B.J. Wielinga [§] **J.M. Akkermans** [‡] **A.Th. Schreiber** [§]

[§]University of Amsterdam, Social Science Informatics
Roetersstraat 15, NL-1018 WB Amsterdam, The Netherlands
E-mail: {wielinga,schreiber}@swi.psy.uva.nl

[‡]University of Twente and ECN
Information Systems Department INF/IS
P.O. Box 217, NL-7500 AE, The Netherlands
E-mail: akkerman@cs.utwente.nl, akkermans@ecn.nl

Abstract

A formal analysis of the problem of parametric design is given on the basis of the competence-theory approach to problem solving methods. In particular, the ontological commitments and assumptions underlying some problem solving methods as used in the VT task are investigated.

1 Introduction

In this paper we explore the formal foundations of methods to solve a class of design problems that require the assignment of values to a set of design parameters (parametric design). Previously, we proposed a general approach that explicates the construction process of problem solving methods (PSMs) employed in knowledge-based systems (Akkermans *et al.*, 1993; Akkermans *et al.*, 1994). As a key point we employed the notion of a competence theory (Van de Velde, 1988) of a problem solving method. Illustrations were taken from the diagnostic Cover-and-Differentiate method used in MOLE (Eshelman, 1988) and from various forms of abductive diagnosis (Schreiber *et al.*, 1992). It is then possible to show how a formal and rational (re)construction of PSMs results from successive conceptual refinement and operationalization steps with respect to the competence theory. Our proposed Competence Theory approach to PSMs intends to provide top-down support for method construction, starting from an informal problem statement to an operational inference structure suitable for knowledge-based reasoning. More generally, it is helpful in analyzing the ontological commitments and background assumptions that a PSM makes with regard to the

[*]The work reported here was partially carried out in the course of the KACTUS project, with a partial funding of the Commission of the European Communities. The partners in this project are: Cap Gemini Innovation (F), LABEIN (Sp), Lloyd's Register (UK), Statoil (N), CAP Programmator (S), University of Amsterdam (NL), University of Karlsruhe (D), IBERDROLA (Sp), DELOS (I), FINCANTIERI (I) and SINTEF (N). This article expresses the opinions of the authors and not necessarily those of the consortia.

domain theory and the required reasoning.

In this paper, we apply the competence theory approach to the problem of parametric design, especially as carried out in the Sisyphus VT task through the Propose-and-Revise PSM (Marcus *et al.*, 1988; Marcus & McDermott, 1989; Yost, 1992), in order to clarify the underlying assumptions and commitments that methods of this type make. In Sec. 2, we first give a brief general survey of the Competence Theory analysis of PSMs. In Sec. 3, we discuss a formal initial competence theory for design and parametric design problems, and in Sec. 4 an analysis is given of the various assumptions underlying some problem solving methods for parametric design.

2 PSM Analysis: The Competence Theory Approach

As discussed in our previous work (Akkermans *et al.*, 1993; Akkermans *et al.*, 1994), the general starting point of the competence-theory approach to PSMs is the view taken by Newell (Newell, 1982) that knowledge is a competence-like notion and that rationality means to apply this competence in order to achieve the given goal, that is, to solve the given problem. A problem solving method can be ascribed a certain competence with respect to the class of problems it has been designed to solve. Therefore, as a key element in our approach we will employ the notion of a *competence theory* of a PSM (cf. (Van de Velde, 1988)): a generic description of the ability that a PSM has to solve a certain class of problems. The rational justification of a PSM lies in demonstrating its competence with respect to the given problem. This is done by outlining how the PSM construction process is guided all along by the conceptual refinement and operationalization of an initial, required, competence theory that is immediately linked to a top-level goal specification.

An overview of the steps in PSM construction and analysis is given in Figure 1:

1. *Specification* of the problem space and of the requirements for the solution. This yields a required competence theory $\mathbf{T}_0$ for the PSM.
2. *Conceptual refinement* of this competence theory introducing the intermediate task and domain vocabulary based on assumptions regarding the available domain theory and the ways the task goal is achieved by the method. This leads to a refined competence theory $\mathbf{T}_1$ of a PSM.
3. *Operationalization* of this refined competence theory to inference structures and associated control regimes that are sufficiently detailed and practical to act as a basis for KBS design. This entails further assumptions of an operational nature and yields an operational specification $\mathbf{T}_2$ of a PSM.

Van de Velde (Van de Velde, 1988) defines the notion of *problem space* as a triple

$$\Omega =< P, Sol, \textit{solution} >, \tag{1}$$

where P is the set of problems in Ω, Sol is the set of solutions in Ω, and *solution* is a relation between P and Sol, that is, *solution* $\subset P \times Sol$. A solution for a specific problem $p \in P$ in

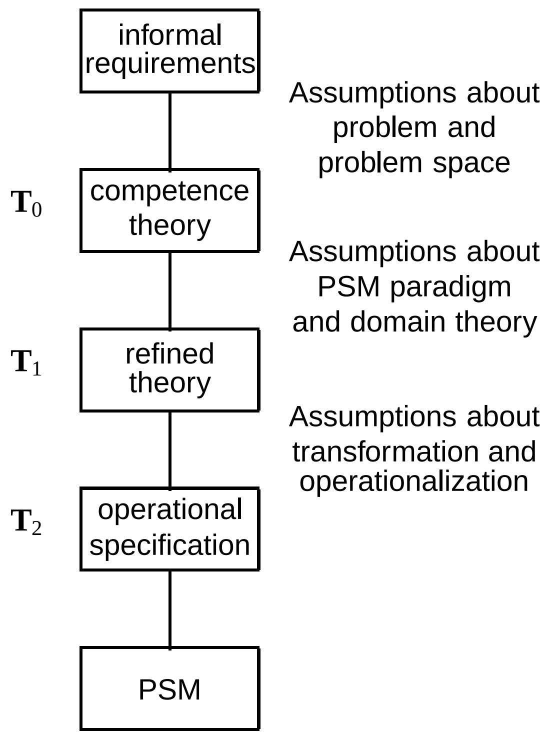
informal requirements
Assumptions about problem and problem space
T_0
competence theory
Assumptions about PSM paradigm and domain theory
T_1
refined theory
Assumptions about transformation and operationalization
T_2
operational specification
PSM

FIGURE 1: Competence Theory approach to PSM analysis and construction.

a problem space Ω then is any $sol \in Sol$ such that *solution(p,sol)* holds. In the ideal case, the solution relation can be fully specified directly; then, the problem solving method boils down to a direct match based on table look-up. But life is usually not that simple, and a further conceptualization of the problem is necessary before a method can be obtained. Based upon the above abstract definitions, problem conceptualization means to analyze in more detail what the ingredients of the problem space Ω are.

3 A Competence Theory for Parametric Design

3.1 The nature of design problems Many design problem can be informally described as a problem in which a set of needs and desires is given and where the solution is a description of some structure that satisfies these needs and desires. Often, the needs and desires are specified in an informal way and leave much unsaid. A client may ask an architect to design a house with a large living room, an American-style kitchen and four bedrooms for a price not exceeding \$ 200,000. Additional requirements —such as the presence of an attic or garage— are often established during a negotiating process between client and designer. The specification of obvious requirements —such as water and electricity— remains implicit. Similarly, external requirements —such as fire and safety regulations, municipal rules, constraints on the use of building materials etc.— are implied but often not stated explicitly. The designer has to perform an *analysis process* that transforms the needs and desires into a more formal and complete set of requirements and constraints. In this analysis process the designer makes use of knowledge of the domain and of external information sources, such as a safety regulations handbook.

Given a more or less formalized set of requirements and constraints the designer will enter a *synthesis process* in which a *structure* is developed —the design— that consists of a number of building blocks or *elements* with specified properties —such as dimensions, material etc.— and relations between them. During the synthesis process, the designer makes use of knowledge about the world and of specific design knowledge, that allows to relate design decisions to the requirements and constraints. The simple model of solving a design problem is summarized in Figure 2. This diagram should not be interpreted as a pure sequential process. Often analysis and synthesis will be performed in an iterative way and requirements and constraints can be constructed in an iterative manner (Smithers *et al.*, 1994). In cases where requirements need to be *constructed* one could argue that the term *analysis* is a misnomer. However, following the KADS tradition we reserve the term *synthesis* for processes that result in some sort of structure.

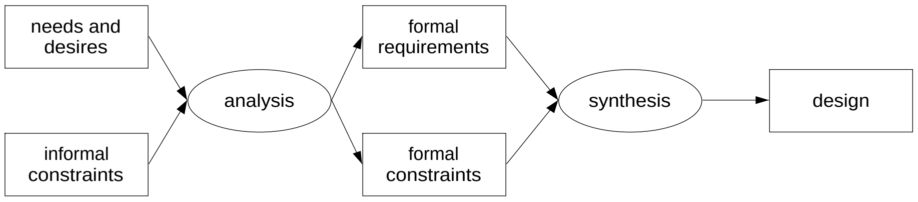

FIGURE 2: Simplistic model of typical design problem solving.

Regrettably, there is no standard(ized) terminology regarding the requirements capture process, neither in knowledge engineering nor in the various fields of design themselves. In the following, we will reserve the term 'requirements' for the wanted or necessary properties that a design must have in order to be acceptable as a solution (Brazier *et al.*, 1994). The term 'constraints' will be used to indicate limitations on what counts as a possible design solution (sometimes, a subset of the constraints is referred to as non-functional requirements). This conforms with the distinctions made by (Tank, 1992), who speaks of requirements proper ("Anforderungen") and boundary conditions ("Randbedingungen"); for an overview of much relevant German work on configuration design, see (Loeckenhoff & Messer, 1994). Other work done in the context of the CommonKADS Expertise Model Library focuses on the above-mentioned early analysis process in design (Bernaras & van de Velde, 1994). Informal needs and desires are constructively developed into requirements, denoting the (qualitative) criteria by which a design solution is to be judged, and are subsequently transformed into a formal and operational problem statement that serves as the starting point of the actual design synthesis process. The emphasis in the DIDS approach to configuration design (Runkel & Birmingham, 1994) is on the latter process. Here, knowledge structures are introduced such as functions of systems parts (which evidently are linked with requirements, but the approach is not concerned with the formalization of this aspect), and constraints which are introduced in the technical sense of algebraic relationships between attributes of system parts.

Apart from the lack of standard terminology, conceptual distinctions that are deemed useful obviously depend on the goals of the work. We therefore do not want to claim a special status for the present terminology. The present work considers the competence of problem solving methods. In this context, the distinction made between requirements and constraints is relevant, because design PSMs will handle them differently. Informally, requirements refer to ‘positive’ properties of the artefact that can be used as handles to cut down the design space considered in the synthesis process —for example, the requirement that a house should have four bedrooms. On the other hand, constraints are ‘negative’ properties in the sense that they have no constructive value in synthesis activities, but are boundary conditions that characterize non-solutions to the design problem —for example, that the total cost of the house may not exceed $ 200,000. We will define the difference between constraints and requirements more formally later. Note furthermore that we employ the term constraint in a conceptual sense, rather than in the technical-mathematical meaning of the term. Both requirements and constraints may, usually somewhere near the end of the analysis process, be formally expressible as mathematical constraint expressions (which is to a large extent indeed the case in the Sisyphus VT task).

In this paper we will restrict ourselves to the synthesis part of the design problem by assuming that the requirements and constraints are already available in an explicit and formal form. A number of explicit *assumptions* for this *formal design* task will be made throughout this paper:

- The problem can be formulated in terms of a set of formal *requirements* and *constraints*.
- The solution can be formulated in terms of a set of *design elements*, a set of *relations* over the elements so that design structures or assemblies can be modelled, and a set of *parameters* that have *values* assigned to them.
- There exists some theory that allows to derive information about the relation between the solution, requirements and constraints (*the Domain Theory*).
- In the design process a number of *design choices* are made which are not part of the domain theory, but which constrain the space of design options. For example, a design choice could be that the set of elements from which a design can be constructed is limited (e.g. Lego blocks) or that values assigned to parameters are initially restricted to preferred values. Design choices may be retracted when no solution can be found within the constrained search space.

3.2 Initial conceptualization of the design problem space The first step in constructing a PSM for the formal design task is to define the problem space in formal way. We define a number of information objects that are relevant in the design process and introduce a notation for them, without specifying at this point what the precise nature of these objects is. Yet, this first conceptualization step provides an initial ontology for what a design problem is.

There are a number of decisions made in the initial ontology shown in Table 1 that are worth discussing. The set of parameters is not restricted to parameters which represent attributes of the *elements* only. We assume that it is possible that certain parameters are not associated with a particular element, but rather with design structures as a whole. For example, the total cost of realizing a design or the production machinery used, does not necessarily correspond to properties of one or more elements of the design structure. Further, we define relations between elements as tuples of a relation name and a number of elements. The name of the relation is explicitly introduced to allow a flexible use of different relations in the problem solving process. The sets

requirement set:	$\{r_i\} = R$
constraint set:	$\{c_i\} = C$
element set:	$\{e_i\} = E$
parameter set:	$\{p_i\} = Par$
value set:	$\forall p_i :\ \{v_{ij}\} = V_i$
assignment set:	$\{a_{ij}\} = \{< p_i, v_{ij} >\} = A$
relation name set:	$\{l_k\} = L$
design structure:	$\{d_{kij} =< l_k, e_i, \cdots, e_j >\} = D$
domain theory:	$\{t_i\} = DT$
requirements partitions:	$R = R_p \cup R_{DT}$
constraints partitions:	$C = C_p \cup C_{DT}$
design choices:	$Z = \{z_i\}$

(2)

TABLE 1: Problem space definitions for design.

of requirements and constraints are partitioned into a part that is given as (explicit) input to the problem solving process and a set of requirements and constraints that are derivable from the *Domain Theory*. The former sets are indicated by R_p and C_p, while the partitions of requirements and constraints derived from the domain theory are indicated by R_{DT} and C_{DT}. We note that this is one way to deal with (a part of) implied but not explicitly stated requirements: input user requirements can leave requirements implicit that can be derived from the underlying theory of the design domain. We assume that the input requirements and constraints together with the domain theory are sufficient to derive the implicit requirement and constraints. The specification of the partitioning of the requirement and constraint sets in the problem space then becomes:

$$R_p \cup DT \cup C_p \vdash R_{DT} \quad \text{and} \quad R_p \cup C_p \cup DT \vdash C_{DT}. \tag{3}$$

This formalizes the notion of unstated but implied requirements and constraints.

problem space:	$\Omega =< P, \mathit{DT}, Z, Sol, \mathit{solution} >$
problem set:	$P = \{p \mid p =< R_p, C_p >\}$
solution set:	$Sol_p = \{sol_p \mid sol_p =< E_p, D_p, A_p > \ \wedge \mathit{solution}(p, sol_p)\}$

(4)

TABLE 2: Problem space specification for design.

The problem space can now be defined in terms of a set of problems –consisting of input requirements and input constraints–, a domain theory, a set of design choices, a solution space and a *solution* predicate, according to Table 2.

Thus, the total design space, *i.e.*, the collection of all conceivable design descriptions that can be generated, is conceptualized as the Cartesian product $E \times D \times A$. Tank calls this space the "Konstruktionenraum" (Tank, 1992) and points out that not all designs are necessarily constructible. The space of possible *constructible* design descriptions is called by Tank the *'Variantenraum'*. We will use the term *realization space*. The realization space is typically constructed on the basis of constraints that represent knowledge about what can be and what cannot be constructed. In the following we will assume that a design description that is consistent with the implicit constraints is constructible. Formally:

$$C_{DT} \cup \mathit{DT} \vdash \quad \exists X : \; X = \{x_i \in E \times D \times A | \mathit{constructible}(x_i)\}, \tag{5}$$

The space of design descriptions that satisfy the requirements is not necessarily contained in the realization space. Certain designs may satisfy the requirements but may not be constructible. Tank calls the space of designs that satisfy the requirements the *satisfaction space* ("Erfullungsraum"). The solution space then is the intersection of the satisfaction space and the realization space.

In addition to the definitions given above, we need to relate the concept of design choices Z to other elements of the problem space. This might be done by introducing a notion concerning preference knowledge in making decisions in the design space. This we do through introducing a predicate *preferred* as follows:

$$Z \cup \mathit{DT} \vdash \quad \exists X : \; X = \{x_i \in E \cup D \cup A | \mathit{preferred}(x_i)\}, \tag{6}$$

which says that there is some theory about preferences in the dimensions that span up the design space, entailed by the domain theory together with the design choices. Preferences introduce yet another subspace of the total design space: preferred design space. Note that this space may have an empty intersection with the solution space.

This completes what we called in (Akkermans *et al.*, 1993) the *problem space description*. What remains is to formulate the solution specification: the formal relationship between the *solution* predicate and the various elements of the problem space. This relationship tells us what it means to be a solution to a problem, and thus specifies the required competence theory of problem solving methods.

In line with the previous discussion on the nature of design problems, we view a design description as a solution to the given design problem, if it can be demonstrated —on the basis of the domain theory and the design description— that all requirements are ('positively') satisfied and ('negatively') that no constraints are being violated. Formally:

$$\mathit{solution}(p, sol_p) \quad \leftrightarrow \quad (sol_p \cup \mathit{DT} \models R) \quad \wedge \quad (sol_p \cup \mathit{DT} \cup C \not\models \perp) \tag{7}$$

This is the *Required Competence* theory of a PSM for solving a formal design problem. Note that the fact that only consistency with the constraints is required, is caused by the possibility that establishing the truth value of some constraints may not be tractable or computable given a particular solution. It may be the case that a solution is found that does not involve certain elements or parameters and that —as a consequence— certain constraints remain undetermined, but consistent.

As pointed out in (Akkermans *et al.*, 1993) in relation to diagnosis problems, one can in general formulate several useful variants of PSM competence theories. In design problems, a possible additional competence statement may be that certain feasible design solutions are considered to be more optimal than others:

$$\neg\exists sol'_p : (sol'_p \prec sol_p) \wedge \textit{solution}(p, sol_p) \wedge \textit{solution}(p, sol'_p), \tag{8}$$

where $\prec$ is some ordering relation (ranking function) based on optimality criteria. In the extended competence theory consisting of Eqs. (7) and (8), Equation (7) yields the feasible or acceptable design solutions, while Equation (8) defines the most preferred solution. The ranking $\prec$ of design solutions is likely to be expressed in relation to the above-mentioned preference knowledge (the *preferred* predicate in Eq. (6)). For example, $\prec$ may be specified as the largest number of preference statements $x_i \in X_p$ for a given design problem, but there are many other possibilities.

3.3 Solution specification for parametric design The required competence theory for design discussed above, needs to be specialized for different forms of design. The VT system (Marcus, 1988; Yost, 1992) is a system that performs parametric design. The Sisyphus VT task is often called a configuration design task. Strictly speaking this is true since not all components of the elevator configuration are fixed. However the way in which the VT task is conceptualized is such that new configurations are represented in a parameterized form. For example the presence of a telephone in the elevator cabin is represented by a parameter that can take the value 0 or 1.

To specialize the competence theory for parametric design, we have to introduce a number of additional assumptions with regard to the problem space:

- The elements (components) and element relations (structures: assemblies of components) are given and predefined.
- Parameters are associated with elements E and structures D.
- A skeletal model that represents the possible set of parameters to which values can be assigned, is part of the domain theory DT.
- Constraints are formulated as algebraic expressions ((in)equalities) over parameters, values and constants.
- Requirements are represented as (initial) assignments of values, or value ranges, to parameters.

These additional assumptions give rise to a further detailing of the problem space, leading to the following problem space and solution specification (9):

$$\begin{array}{rcl} p & = & < R_p, C_p > \\ R_p \cup C_p & \vdash & A_p \\ DT & \supset & E_p \cup D_p \\ E_p \cup DT \cup D_p & \vdash & Par_p \\ Z & \vdash & A_{init} \\ sol_p & = & A_{sol} \supset A_p \\ \textit{solution}(p, sol_p) & \leftrightarrow & (sol_p \cup \textit{DT} \models R_p) \wedge (sol_p \cup \textit{DT} \cup C_p \not\vdash \bot) \end{array} \tag{9}$$

TABLE 3: The competence theory for parametric design (T_0)

Characteristic for parametric design is that essentially all relevant knowledge is expressed as mathematical constraint expressions over design parameters. A parametric design problem is given by an input set of requirements and constraints, but such that these can be converted into a set of value or value range assignments A_p of parameters. Next, the domain theory, which includes knowledge about design elements and structures (design space), must enable to derive the set of parameters Par_p that is relevant for the given problem. Furthermore, from the design choices Z it is possible to derive a theory of initial assignments A_{init}; these may be seen as (proposed) 'soft' assignments whereas A_p covers the demanded 'hard' assignments. The design solution sol_p must in parametric design also be expressible as a set of assignments A_{sol} that has to cover the hard requirements and constraints A_p as a superset. We assume that the solution specification Eq. (7) remains valid: all requirements are satisfied, and no constraints may be violated.

The theory (9) is the *Required Competence Theory* $\mathbf{T}_0$ of parametric design. It lays down the requirements for problem solving methods like Propose-and-Revise. These are declarative requirements related to the PSM competence, that is, they tell us which class of problems the PSM can solve. Conceivably, at a later stage efficiency requirements may be added, *e.g.*, that $A_{init} \cap A_{sol}$ must be as large as possible in order to generate good proposals that minimize backtracking and fixing.

4 The Refined Competence Theory for Parametric Design

Having defined the required competence theory (T_0) for parametric design we must further refine this theory such that the statements in the competence theory can be related to the content of the domain theory. Such a refinement reveals how certain derivations that remain implicit in T_0 can actually be achieved using statements in the domain theory. This refinement involves four steps:

- Introduction of a number of predicates that allow the representation of the domain knowledge in an explicit form.
- Selection of a *Problem Solving Paradigm*, that indicates how the high-level specification of the solution predicate will be further decomposed.

- Refinement of the "$\models$" operator into "normal" logical inference steps.
- Expansions of the set notation into quantified expressions.

4.1 Domain theory assumptions We assume that the following assumptions hold for DT:

- The set of possible parameters is represented through the predicate *parameter(p_i)*.
- For each parameter p_i a predicate *used_in(p_i,x_j)* indicates in which requirements or constraints the parameter occurs.
- DT contains predicates of the form *meets(A_i,r_i)* that indicate that a set of value assignments satisfies a requirement.
- The domain theory contains a set of *formulae* that can be used to calculate values of parameters using already known values (*calculation(p_i,a_i)*). Note that the VT ontology calls these expressions constraints.

4.2 Refining the Competence (1) We now need to define the way in which the *solution* relation is derived from the knowledge base (Theory 1). In a first refinement step we need to specify how the parameter space is determined.

In the VT task it is not necessary that all parameters get assigned a value. The relevance of a parameter can depend on certain assignments of other parameters. For example, several parameters depend on the model that is chosen for a component. We define the set of parameters that are relevant given a particular solution as follows:

$$\begin{aligned} Par_s = \{p_i | parameter(p_i) \wedge (\text{used\_in}(p_i, R_p) \vee \text{used\_in}(p_i, C_p)) \vee \\ \forall p_j \neq p_i, x \ \text{used\_in}(p_i, x) \wedge \text{used\_in}(p_j, x) \wedge p_j \in Par_s\} \end{aligned} \tag{10}$$

This definition says that a parameter is relevant if it is used in one of the input requirements or input constraints, or if it occurs in a requirement or constraint (indicated by x) in which another relevant parameter occurs.

The fact that the definition of the set of relevant parameters is recursive reflects the complex nature of the dependencies between parameters. One underlying assumption is that initial requirements and constraints defined by the user as part of the problem statement, specify an initial subset of the relevant parameters. This assumption is not unreasonable but may not always hold. The second, recursive, part of the definition of the relevant parameter set is based on the assumption that the relevance property propagates through the *used_in* relation. This can cause difficulties in cases where the network of dependencies contains mutually recursive cycles.

The set of relevant parameters is needed when we want to convert the equations that define the *solution* predicate, to quantified expressions. However before we can do so, we have to select a problem solving paradigm that describes how assignments come about in the first place. The *Generate-and Test* paradigm is an obvious first choice.

In the decomposition of the competence theory of parametric design in a generate and a test part, two options exist: we can first generate a complete set of assignments to relevant parameters and subsequently test that assignment, or we can generate a partial assignment (e.g. one value for a single parameter) and test the consistency with the constraints and the satisfaction of the requirements. The first method is similar to the "complete-model-then-revise" model described in (Motta *et al.*, 1994), while the second resembles the "extend-design-then-revise" model which is the basis of many operationalizations of the Propose-and-Revise method (Motta *et al.*, 1994), (Schreiber *et al.*, 1994), (Puppe, 1993).

We will refine the competence following the first option: first generate a complete model and subsequently test it. The generation of an assignment is assumed to be done in an abductive fashion: parameter values are proposed on the basis of the *meets* predicate or on the basis of calculations. We could introduce a *propose1* predicate defined as follows:

$$\textit{propose1(p,a)} \leftarrow (\exists A_r \textit{meets}(A_r,r), a \in A_r) \vee \textit{calculation(p,a)} \tag{11}$$

However, the problem with this definition is that for a calculation to produce a value, usually other assignments are needed. A better, but more complex definition of *propose* is a recursive one:

$$\textit{propose\_value}(p,a,A) \leftarrow (\exists A_r \textit{meets}(A_r,r), a \in A_r) \vee (\textit{propose}(A) \ \wedge \ \textit{calculation}(p,a,A)) \tag{12}$$

$$\textit{propose}(A_{proposed}) \leftarrow A_{proposed} = \ \{a_i \mid \forall p_i \in Par_s \ \textit{propose\_value}(p_i, a_i, A_{proposed})\} \tag{13}$$

This definition states that a value assignment can be generated abductively using the *meets* predicate assumed to be present in DT, or that a value can be calculated using the formulae in DT and a set of proposed assignments. There are several observations regarding this formula that are worth noting. First, there is no guarantee that an assignment exists that covers all parameters in Par_s. The set of requirement specifications could be incomplete, the set of calculations could be incomplete or certain cyclic dependencies between parameters can exist that prevent the derivation of a calculation. This problem can be solved by adding a term to the definition of *propose_value* that will select an assignment from the range of possible values. This would guarantee that propose will ultimately generate the entire design space.

Secondly, there is no guarantee that the generated assignment satisfies the requirements. Not only the abductive use of the *meets* predicate, but also the calculations do not imply that requirements are met. If we adhere to the assumption discussed in section 4.1, that requirements are formulated as required values for parameters, even then there is no guarantee that requirements generated from the domain theory will be satisfied.

This problem can be solved by performing a proper test on the proposed assignment with respect to the satisfaction of the requirements. Interestingly enough, many of the VT systems that were presented during Banff'94 implicitly assume that proposed assignments satisfy the requirements and consequently perform no satisfaction test.

A third observation concerns the role of preferences. The definition of *propose* does not involve preferred values for assignments. Within the context of the *Propose-and Revise* paradigm the space of generated assignments would be restricted to the preferred design space. This can however only be done if changes are allowed to a proposed assignment, since there is no guarantee that the preferred space contains a viable solution. This is impossible within the *Generate-and-Test* paradigm.

The test part of the refined competence theory will contain two tests: a consistency check for the constraints and a satisfaction test for the requirements. First we define a predicate *consistent* which is true if no inconsistency can be proven for a particular constraint given a particular assignment:

$$\textit{consistent(c,A)} \leftarrow \neg \textit{prove}(\neg c, A) \tag{14}$$

Assuming that the assignment of values to parameters are not inconsistent with the domain theory and that the set of constraints is internally consistent, we can rewrite the consistency requirements of the competence theory as follows:

$$(sol_p = A_s \cup DT \cup C \not\vdash \bot) \leftrightarrow \forall c \in C \;\; consistent(c, A_s) \tag{15}$$

In a similar spirit we introduce a predicate *test* that tests whether a requirement is met by a particular assignment, and we relate this predicate to the satisfaction of the set of requirements:

$$test(r, A) \leftarrow \exists A_r \subset A \;\; meets(A_r, r) \tag{16}$$

$$(Sol_p = A_s \cup DT \models R) \leftrightarrow \forall r \in R \;\; test(r, A_s) \tag{17}$$

We are now in a position to formulate the refined competence theory (T_1) for parametric design according to the *Propose-complete-design-and-Test* method.

$$\begin{aligned} \textit{solution(p,}Par_s\textit{,A)} \leftrightarrow \textit{propose(}Par_s\textit{,A)} \wedge \\ \textit{consistent(A)} \wedge \\ \forall r \in R_p \; test(r, A) \end{aligned} \tag{18}$$

TABLE 4: Refined competence theory of the Propose-complete-design-and-Test method

It is not difficult to see how this competence theory could be transformed into a KADS-like model for an operational system. A simple backtracking control mechanism would suffice to realize such a system.

Note that the order in which we introduce the decomposition and the expansion of sets into quantified expressions, has a significant impact on the refinement and the subsequent operationalization. In particular the control component of the final knowledge model will be rather different if we chose to decompose first and then introduce quantification, from when we do this the other way around. Also assumptions about efficiency, locality of operations etc. play a role in this decision.

The importance of this exercise is that it reveals the extent of the assumptions that have to be made in order to arrive at such a simple model. Would we have chosen the second option outlined above, different assumptions would have been needed.

4.3 Refining the competence (2) The refined competence theory specified in equation 18 can be operationalized as a search process through the space of possible value assignments. This PSM is potentially very inefficient. The design space can be very large and a simple search process may become computationally intractable. The *Propose-and-Revise* method reduces the search space to a subspace that is defined by a number of propose heuristics and "fixes".

The *Propose-and Revise* method is based on the idea that the initial proposal for an assignment can be constructed by selecting value assignments from the preferred value space, and that this proposed assignment can be "fixed" if constraints are violated. So, the propose step only searches the space of preferred value assignments, while the fixes define a larger space of value assignments that is traversed when violations of the constraints occur.

First we define the restricted space of value assignments that is generated by the propose step.

$$\textit{propose\_value}(Par_s,p,a,A) \leftarrow preferred(p,a) \vee \tag{19}$$

$$(\text{propose}(Par_s, A) \ \wedge \ \textit{calculation(p,a,A)}) \tag{20}$$

$$propose(Par_s, A_{proposed}) \leftarrow \ \exists A_{proposed} = \tag{21}$$

$$\{a_i | \forall p_i \in Par_s \ \text{propose\_value}(Par_s, p_i, a_i, A_{proposed})\} \tag{22}$$

Using predicates consistent and test similar to the ones defined above and introducing a predicate fix that transforms an assignment to another assignment we can define the refined competence for the method *Propose-complete-design-and Revise* as follows:

$$\begin{aligned} \textit{solution}(p,Par_s,A_s) \leftrightarrow \textit{propose}(Par_s,A) \wedge ((\ \textit{consistent}(A) \rightarrow A = A_s) \vee \\ (\neg\ \textit{consistent}(A) \wedge fix(A, A_s))) \wedge \\ \forall r \in R \ \ test(r, A_s)) \end{aligned} \tag{23}$$

TABLE 5: Refined competence theory of the Propose-complete-design-and-Revise method

Although this theory captures the intuitions behind the propose and revise method it hides several complexities in the predicates *consistent* and *fix*. The fixing of a proposed assignment when a constraint is violated introduces a non- monotonic element in the competence theory which is difficult to capture in first order logic. What would be needed is a complex recursive definition of the assignment set and an account of how fixes propagate through the network of parameter dependencies. Although we have attempted to construct such competence theories, their understandability is greatly reduced when the non-monotonic aspects of fixes are properly accounted for. In addition it becomes very difficult to formalize the precise assumptions under which a propose and revise solution satisfies the initial competence theory formulated in section 3. The conclusion that we draw from this is that the partial formalization of the *Propose-and-Revise* method already reveals a large set of assumptions and ontological commitments, but that more work needs to be done before we can formally prove under what conditions such methods yield a viable solution to a parametric design problem.

5 Conclusions

In this paper we have developed a formal account of the synthesis part of design problems in general and of parametric design problems in particular. Starting with a set of definitions we have developed a *required competence theory* that formally specifies the relation between a design problem and a solution. Subsequently we have introduced a number of assumptions that lead to the specification of the required competence of problem solving methods for parametric design. Even at this high level of abstraction, the analysis shows what the fundamental notions and ontological commitments are in tasks such as the VT task of designing elevator configurations. In particular, our analysis shows the differences between requirements, constraints and preferences in the design process.

The power of the approach sketched in this paper is that it explicates the ontological commitments and assumptions behind the problem solving methods that are employed in systems that solve a particular class of tasks. The various solutions to the VT problem can be viewed as different refinements and operationalizations of the initial competence theory. The explication of these commitments and assumptions places strong constraints on the form and content of the knowledge represented in the domain theory. As such, the formalization process outlined in this paper also poses strong constraints on the knowledge acquisition process. For example, the assumptions explicated in this paper can be viewed as a formal basis for techniques that construct models of reasoning processes through a refinement process, such as the GDM approach (van Heijst *et al.*, 1992).

The formalization process developed in this paper is not without problems. The non-monotonic nature of the *Propose and Revise* method is difficult to capture in intuitively understandable theories. We are convinced however, that further work in formalization of PSMs will ultimately lead to a better understanding of what the competence of Knowledge Based Systems really is. Developing problem solving methods and systems that empirically show their heuristic value is one thing, but ultimately the scientific goal of the field of knowledge engineering should be to understand what problems these systems can solve and what problems they cannot solve.

In this paper we have barely touched upon the problem of transforming the refined competence theory into an operational knowledge model that can be implemented. This is beyond the scope of the present paper, but will be the focus of future work.

References

AKKERMANS, J. M., WIELINGA, B. J., & SCHREIBER, A. T. (1993). Steps in constructing problem-solving methods. In Aussenac, N., Boy, G., Gaines, B., Linster, M., Ganascia, J.-G., & Kodratoff, Y., editors, *Knowledge Acquisition for Knowledge-Based Systems. Proceedings of the 7th European Workshop EKAW'93, Toulouse and Caylus, France*, number 723 in Lecture Notes in Computer Science, pages 45–65, Berlin Heidelberg, Germany. Springer-Verlag.

AKKERMANS, J. M., WIELINGA, B. J., & SCHREIBER, A. T. (1994). Steps in constructing problem-solving methods. In Gaines, B. R. & Musen, M. A., editors, *Proceedings of the 8th Banff Knowledge Acquisition for Knowledge-Based Systems Workshop. Volume 2: Shareable and Reusable Problem-Solving Methods*, pages 29–1 – 29–21, Alberta, Canada. SRDG Publications, University of Calgary.

BERNARAS, A. & VAN DE VELDE, W. (1994). Design. In Breuker, J. & van de Velde, W., editors, *CommonKADS Library for Expertise Modelling*, pages 175–196. Amsterdam, IOS press.

BRAZIER, F., VAN LANGEN, P., RUTTKAY, Z., & TREUR, J. (1994). On formal specification of design tasks. In Gero, J. & Sudweeks, F., editors, *Artificial Intelligence in Design '94*, pages 535–552. Dordrecht, Kluwer Academic Publishers.

ESHELMAN, L. (1988). MOLE: A knowledge-acquisition tool for cover-and-differentiate systems. In Marcus, S., editor, *Automating Knowledge Acquisition for Expert Systems*, pages 37–80. Boston, Kluwer.

LOECKENHOFF, C. & MESSER, T. (1994). Configuration. In Breuker, J. & van de Velde, W., editors, *CommonKADS Library for Expertise Modelling*, pages 197–212. Amsterdam, IOS press.

MARCUS, S., editor (1988). *Automatic knowledge acquisition for expert systems*. Boston, Kluwer.

MARCUS, S. & MCDERMOTT, J. (1989). SALT: A knowledge acquisition language for propose-and-revise systems. *Artificial Intelligence*, 39(1):1–38.

MARCUS, S., STOUT, J., & MCDERMOTT, J. (1988). VT: An expert elevator designer that uses knowledge-based backtracking. *AI Magazine*, Spring:95–111.

MOTTA, E., OHARA, K., N. R. SHADBOLT, A. S., & ZDRAHAL, Z. (1994). A vital solution to the sisyphus ii elevator design problem. In Schreiber, A. T. & Birmingham, W. P., editors, *Proceedings of the 8th Banff Knowledge Acquisition for Knowledge-Based Systems Workshop. Volume 3: Sisphys-II VT Elevator Design Problem*, pages 40–1, Alberta, Canada. University of Calgary, SRDG Publications.

NEWELL, A. (1982). The knowledge level. *Artificial Intelligence*, 18:87–127.

PUPPE, F. (1993). *Systematic Introduction to Expert Systems*. Berlin, Springer-Verlag. ISBN 3-540-56255-9.

RUNKEL, J. T. & BIRMINGHAM, W. P. (1994). Solving vt by reuse. In Schreiber, A. T. & Birmingham, W. P., editors, *Proceedings of the 8th Banff Knowledge Acquisition for Knowledge-Based Systems Workshop. Volume 3: Sisphys-II VT Elevator Design Problem*, pages 42–1, Alberta, Canada. University of Calgary, SRDG Publications.

SCHREIBER, A. T., TERPSTRA, P., MAGNI, P., & VAN VELZEN, M. (1994). Analysing and implementing VT using COMMON-KADS. In Schreiber, A. T. & Birmingham, W. P., editors, *Proceedings of the 8th Banff Knowledge Acquisition for Knowledge-Based Systems Workshop. Volume 3: Sisyphus II – VT Elevator Design Problem*, pages 44–1 – 44–29, Alberta, Canada. SRDG Publications, University of Calgary.

SCHREIBER, A. T., WIELINGA, B. J., & AKKERMANS, J. M. (1992). Differentiating problem solving methods. In Wetter, T., Althoff, K.-D., Boose, J., Gaines, B., Linster, M., & Schmalhofer, F., editors, *Current Developments in Knowledge Acquisition - EKAW'92*, pages 95–111, Berlin, Germany. Springer-Verlag.

SMITHERS, T., CORNE, D., & ROSS, P. (1994). On computing exploration and solving design problems. In *Proceedings of the International Workshop on Formal Methods for CAD*. North Holland.

TANK, W. (1992). *Modellierung von Expertise über Konfigurierungsaufgaben*. Sankt Augustin, Germany, Infix. ISBN 3-929037-05-X.

VAN DE VELDE, W. (1988). Inference stucture as a basis for problem solving. In Kodratoff, Y., editor, *Proceedings of the 8th European Conference on Artificial Intelligence*, pages 202–207, London. Pitman.

VAN HEIJST, G., TERPSTRA, P., WIELINGA, B., & SHADBOLT, N. (1992). Generalised directive models. In *Proceedings of KAW-92, Banff*.

YOST, G. (1992). Configuring elevator systems. Technical report, Digital Equipment Corporation, 111 Locke Drive (LMO2/K11), Marlboro MA 02172.