Int. J. Human–Computer Studies (1996) **44,** 373–402

Sisyphus-VT: A CommonKADS solution

A. Th. Schreiber and P. Terpstra

University of Amsterdam, Social Science Informatics, Roetersstraat 15, NL-1018 WNB Amsterdam, The Netherlands. email: schreiber@swi.psy.uva.nl

This article represents a CommonKADS contribution to the Sisyphus-VT experiment. This experiment is concerned with a comparison of knowledge modelling approaches. The data set for this experiment describes the knowledge for designing an elevator system. The ultimate goal is to arrive at standards for sharing and reusing problem-solving methods and related ontologies.

© 1996 Academic Press Limited

1. Knowledge modelling approach

This article contains a CommonKADS description of the Sisyphus '93 problem: the VT elevator design. The term "CommonKADS" is used to refer to KADS as it has been developed within ESPRIT project P5248 KADS-II. The goal of this project was to arrive at a *de facto* European standard for KBS development. As the term "KADS" has become strongly associated with the University of Amsterdam, it is worthwhile to point out that CommonKADS is the result of a cooperative effort of a number of partners, in particular also (in the context of this paper) the Free University of Brussels and the Netherlands Energy Research Centre ECN.

A central theme of CommonKADS is the idea that various perspectives or *models* are important in the KBS development process: models of the organization, of the overall task, of the agents involved, of the required communication, of the expertise, and of the design of the final artifact. In this article the focus is on expertise modelling and its relation to design and implementation. An overview of CommonKADS can be found in Schreiber, Wielinga, de Hoog, Akkermans and Van de Velde (1994*b*); the textual and graphical notations used are described in Schreiber, Wielinga, Akkermans, Van de Velde and Anjewierden (1994a).†

In Section 2 a brief characterization is given of the VT data set and of the application task. Section 3 describes the structure of the VT domain knowledge in the form of two ontologies. In Section 4 a top-down description is given of tasks and inferences involved. The methods underlying this decomposition are characterized. System design and implementation are addressed in Section 5. The process of using the VT domain theory within our application is discussed also in this context. Section 6 discusses the possible contributions of this paper, as well as its limitations. This article assumes that the reader is familiar with the VT data set and the associated ONTOLINGUA ontologies.

† See also the WWW pages at http://www.swi.psy.uva.nl/projects/CommonKADS/home.html.

1071-5819/96/3/40373 + 30$18.00/0 © 1996 Academic Press Limited

2. Initial problem analysis

2.1. NATURE OF THE VT DATA SET

The emphasis in the VT application description lies on the domain knowledge. This domain description is biased in the sense that it contains precisely the knowledge necessary for the propose-and-revise (P&R) method (Marcus, Stout & McDermott, 1988). At numerous places in the text indications are given of how the knowledge should be applied in a particular step of P&R. Therefore, we decided to model and implement the VT task using the P&R method. Given the goals of Sisyphus (comparison of knowledge modelling approaches) this is a logical choice. We do not claim that P&R is necessarily the best method to solve the VT problem.

2.2. CHARACTERIZATION OF P&R AS A METHOD FOR SOLVING DESIGN PROBLEMS

Informally, a (routine) design task can be characterized as follows (Chandrasekaran, 1988).

- There exists a fixed set of potential components of the artifact to be designed. No creative design of components is performed.
- The goal of design is twofold, namely
  (1) to select and arrange a set of components, and
  (2) to specify values for component characteristics (i.e. parameters) in such a way that the result meets the requirements and satisfies internal constraints.

Tank (1992) defines two specializations of this routine design task, each focusing on one of the two goals.

- *Arrangement* (or *layout*) is a specialization of design in which the components have fixed parameter values, and the design problem is restricted to the selection and arrangement of components. An example of this type of configuration is the configuration of plastic parts on a mould (Barthèlemy, Frot & Simonin 1988).
- *Parametric design* is a specialization of design in which the arrangement of components is fixed. The design problem is in this case limited to finding values for component parameters.

Within this typology P&R can be classified as a method that supports a restricted form of design, namely parametric design problems. The method does not require an explicit structural model of the artifact: the artifact is represented as a flat set of parameters. All arrangement problems have to be reformulated as parametric design problems in order to be solvable by P&R. One consequence is that the method itself is unable to explain the result of its reasoning in terms of a structural device model.

3. Domain-knowledge description

A CommonKADS domain-knowledge description consists of three parts.

Ontologies: sets of type definitions of domain constructs, such as concepts and relations.
Ontology mappings: a description of how types defined in one ontology can be mapped onto types in another ontology.

Domain models: potential knowledge base partitions containing domain expressions that use a set of ontology definitions.

The rationale for having multiple ontologies, and explicit mappings between those ontologies, lies in the CommonKADS "relative interaction" hypothesis (Schreiber *et al.,* 1994*b*): different types of knowledge differ in the degree in which they are dependent on the nature of the task. This hypothesis is put into use as a principle for organizing the domain knowledge. Ontologies are a means to group specifications of conceptualizations that share the same level of domain, task or method dependency.

Based on this principle, we distinguished two ontologies in the VT case.

(1) *Parametric design ontology.* This ontology is a *task-type oriented* ontology describing general ontological commitments in the context of the parametric design task. It is assumed not to be biased towards a particular *method* for solving a parametric design problem.

(2) *Propose-and-Revise ontology.* The P&R ontology is a *method-oriented* ontology describing ontological commitments specific for the method selected to solve a parametric design task.

The P&R ontology is thus more specific (application-dependent) than the parametric-design ontology. Each of the two ontologies has one or more corresponding domain models. Also, a (partial) mapping is specified between the two ontologies. Figure 1 gives an overview of the ontologies, ontology mappings and domain models used in VT. We discuss the ontologies and the mapping in more detail in the rest of this section.

We used the CommonKADS CML (Conceptual Modelling Language) (Schreiber *et al.,* 1994*a*) to describe the VT domain knowledge. Compared to ONTOLINGUA, the CML notation provides a more expressive but less formal vocabulary for

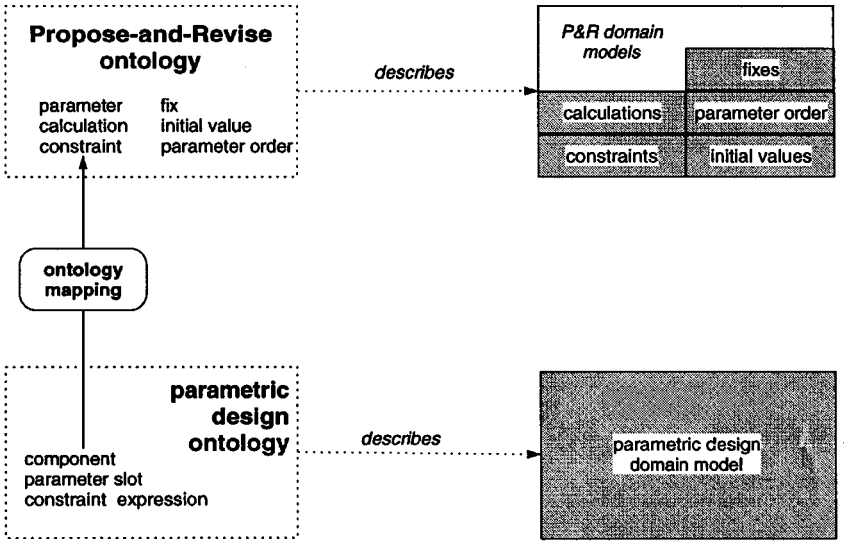


FIGURE 1. Ontologies, ontology mapping and domain models in VT.

specifying ontologies. CML is probably best viewed as a tool for generating an initial semi-formal description of an ontology, which could be refined at a later stage to an ONTOLINGUA-like formal specification. Within the framework of CommonKADS a formal language $(ML)^2$) was developed for knowledge-level specifications together with a number of tools for transforming CML descriptions into skeletal formal specifications (van Harmelen & Balder, 1992; Aben, 1995).

3.1. TASK-TYPE ORIENTED ONTOLOGY: PARAMETRIC DESIGN

We interpreted the ONTOLINGUA specification of the ontology underlying the VT domain as a proposal for a task-type oriented ontology. In the original ''description'' slot of the `configuration-design` ontology it was indicated that the purpose of the ontology was to provide the minimal ontological commitments for solving this task. The authors expected that applications would define their own specific interpretations of this basic knowledge. Also, knowledge about fixes was considered to be P&R-specific and not always required in the general case, and was therefore not included in the ONTOLINGUA definitions.†

Figure 2 shows how the parametric-design ontology is graphically represented in the CommonKADS CML. Boxes represent concepts (classes); diamonds represent relations. Binary relations may have a directional arrow, when the relation is asymmetric. A constraint is modelled as a concept representing a reification (a name) of a formula. The box labelled `parameter-slot` is a special type of concept called an *attribute,* which denotes a concept pointing to a value. The oval represents an *expression,* a modelling primitive in the CML. This construct is used to model simple expressions about values such as $p = 10.5$. It is used here to model expressions about the value of a `parametric-slot`. The two binary relations `fixed-model-value` and `fixed-component-value` link component models (or

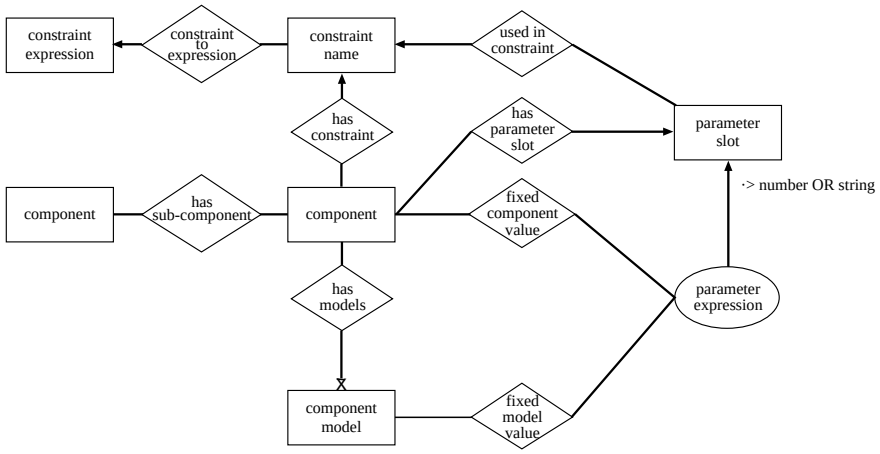


FIGURE 2. Parametric-design ontology. Boxes represent concepts (classes); diamonds represent relations. The oval represents a reified function (a modelling primitive in the CML). The dashed arrow extending from the `has-sub-component` relation represents a property (i.e. function) defined on the relation.

† During the Sisyphus discussions it has been argued that fix knowledge is an integral part of all design problem solving, but that is another issue.

components) to expressions about fixed values of their parameters, e.g. *fixed-model-value*(*safety-B4, safety-beam-height* = 10).

Although there is a large overlap between the ONTOLINGUA ontology and this one, there are a few differences. These are of three types.

(1) *Terminology.* We use the phrase ''constraint name'' instead of ''constraint'' to avoid confusion between the label given to a constraint expression (which is what ''constraint'' is used for in the ONTOLINGUA specification) and the actual expression.
(2) *Representation.* Due to the different representational primitives underlying ONTOLINGUA and CML, some constructs are represented differently. For example, in the ONTOLINGUA VT domain theory the fixed values are represented using the GF (generic frame) formalism, whereas in CML these have to be represented explicitly using a relation construct.
(3) *Extensions.* The CommonKADS parametric-design ontology contains one major extension, namely the distinction between `component` and `component-model`.

In the ONTOLINGUA definition no explicit distinction is made between a component such as `platform` and a particular type of platform, e.g. `platform-4B`. Both are represented as components. However, in design domains it is useful to make a distinction between a ''component'' and a ''component model''. `Component` is used to refer to the general category, i.e. `platform, safety`, etc. A `component-model` refers to the actual type of component that is selected during design: `platform-2B, safety-B4`. Component models are typically associated with particular fixed parameter values, e.g. `safety-B4` always has a height of 10 inches. The `has-models` relation defines how components and component models are related. Component models inherit the parameter slots defined on components, but may introduce additional slots specific for a component model (cf. the parameter slot *x* and *s* of a platform).

In an implicit manner, the ONTOLINGUA definition actually makes a distinction between components and their models (see the VT domain theory).

- Direct sub-classes of `vt-component` are components.
- Direct sub-classes of components are component models.
- The `model-if` slot fulfills a special role by linking components and their models.

We have made this distinction explicit through these two different concept classes. This is warranted by the fact that these two concepts refer to different entities with different characteristics. Representing these through class/sub-class or class/instance distinctions does not capture these differences adequately. Runkel, Birmingham and Balkany (this issue) make a similar distinction between *function* (i.e. component) and *part* (i.e. component model).

3.2. METHOD-ORIENTED ONTOLOGY: P&R

The P&R ontology describes the specific conceptualizations that are required for the P&R problem solving method. We distinguished two parts of this method-oriented ontology based on their different ''reusability status''.

(1) One part of the P&R ontology can be defined as a *method-specific viewpoint* on the parametric-design ontology. This concerns, for example, distinctions that can be made within the set of constraint expressions.
(2) Another part of the P&R ontology contains additional conceptualizations that fall outside the scope of the parametric-design ontology. An example in VT is the fix knowledge.

We discuss these two parts separately.

(1) **Viewpoints on the parametric-design ontology.** The parametric-design ontology defines the minimal conceptualizations required for this task. The assumption is that for a particular method-specific viewpoints can be defined on elements of this ontology which reflect their use in reasoning. The main constructs in this part of the P&R ontology are:
(i) *Parameters.* P&R assumes that a design is represented as a flat set of parameters. This means that components for which during design a component model has to be chosen need to be represented as parameters. Parameter slots are also represented as parameters.
(ii) *Calculations and constraints.* The P&R method also partitions the set of constraint expressions into two sub-sets that each play a different role in the reasoning process: `calculations` and `constraints`. Calculations contain a `mathematical-expression`: a formula that produces a value for a parameter. Constraints contain a `logical-expression` that evaluates to either true or false.

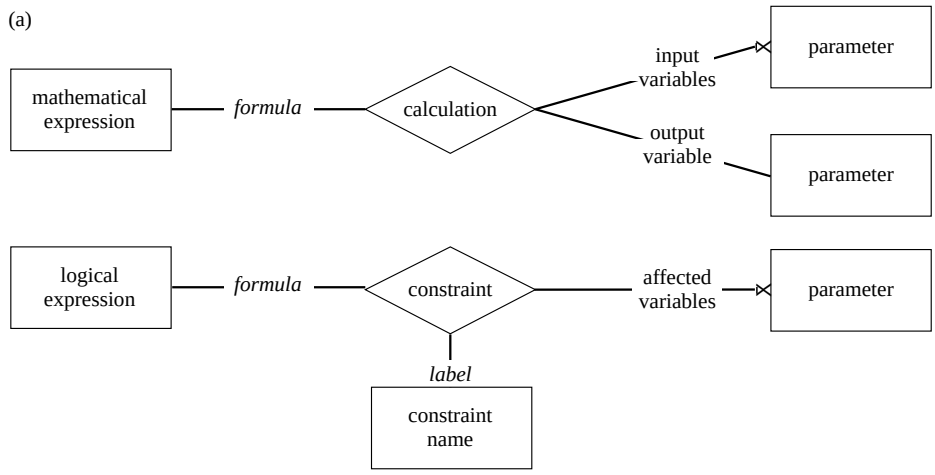
Figure 3(a) shows the graphical CML representation of this part of the P&R ontology. Both calculations and constraints are represented as ternary relations. The italic annotations denote argument roles. Figure 3(b) shows a number of example tuples of the two relations. The formula typically corresponds to a constraint expression or some fixed value in the parametric-design ontology. The examples also show that the P&R representation is redundant: the information about input/output variables could have been derived from the formula. This is typical for a method-specific representation, as the redundant information is useful for efficient reasoning.

The ontology mapping defines how the viewpoint can be realized. An informal description of the mapping between the parametric-design ontology and the P&R ontology is shown in Figure 4. The mapping is a partial one: not all constructs in the parametric design ontology are mapped onto constructs in the P&R ontology. For a formal description a rewrite formalism such as used in $(ML)^2$ (van Harmelen & Balder, 1992) can be used.

The mappings were used in the VT implementation to reuse the VT domain theory within our application. Figure 5 shows the actual results of the transformation of expressions in the parametric-design ontology to expressions in the P&R ontology. The details of this transformation process are described in Section 5.

The P&R-specific distinction between `calculation` and `constraint` implies that only a sub-set of the constraint-expressions in the parametric-design ontology plays the role of ''constraint'' during problem solving, meaning that violation of the constraint expression gives rise to some repair action. Thus, for the P&R method

(a)



(b)

```
calculation(formula(machine-total-weigh = machine-weight + motor-weight)
            input-variables({machine-weight, motor-weight})
            output-variables(machine-total-weight))

calculation(formula(car-speed = 200 => hoist-cable-safety-factor-min = 8.75)
            input-variables({car speed})
            output-variables (hoist-cable-safety-factor-min))

calculation(formula(safety-beam-model=safety-b4 => safety-beam-height = 10)
            input-variables({safety-beam-model})
            output-variables(safety-beam-model))

constraint(formula(2 < platform-weight-factor-ap < 11)
            affected-variables({platform-weight-factor-ap})
            label(platform-weight-factor-ap-c01))
```

FIGURE 3. (a) CML representation of the two relations `calculation` and `constraint` in the P&R ontology that can be expressed as a viewpoint on the parametric-design ontology, (b) some sample tuples of the two relations.

only a sub-set of the domain constraints are ‘‘real’’ constraints; the others are only used to compute values. In this way, a method introduces its own vocabulary, independent of the underlying task-type oriented ontology.

(2) **Additional method-specific ontological commitments.** The method-oriented ontology contains usually also additional conceptualizations introduced by a particular method. The prominent example in the VT domain is the knowledge concerning fixes. Another example of method-specific knowledge is contained in the statement at the beginning of Section 5 of the VT description, namely that the ordering of components in this section can be used as a rough guide for the order in which values should be proposed by the `propose` step. Table 1 lists the additional P&R-specific conceptualizations.

```
ONTOLOGY-MAPPING
  FROM: parametric-design-ontology;
  TO: P-and-R-ontology;
  MAPPINGS:
  " parametric-design        P-and-R-ontology
   ------------------------------------------------------------------------
    parameter-slot         |-> parameter
    component              |-> parameter
                               IF the component has associated component
                                 models
    component-model        |->
                             parameter-value
    constraint-expression  |-> calculation
                               IF the expression can be classified as a
                                 mathematical-expression.
                               Typical format of a mathematical-expression:
                                      <par> = <expression>
                                  OR  <expression> -> <par> = <expression>
    constraint-expression  |-> constraint
                               IF the expression can be classified as a
                                 logical-expression
    fixed-model-value      |-> calculation
    fixed-component-value  |-> calculation                          ";
END ONTOLOGY-MAPPING
```

FIGURE 4. Informal specification of mappings from types in the parametric design ontology onto types in the P&R ontology.

4. Problem-solving model

The aim of this section is to construct a CommonKADS description of the P&R method. Tasks and inferences are described in a top-down fashion.

4.1. PARAMETRIC DESIGN

The top-level task that we used as the starting point for expertise modelling is *parametric design.* In CommonKADS the identification of this task would be the result of building a *task model.* In this model tasks are identified in the current and/or future organization. Expertise modelling is done for those tasks in the task model that have been assigned to a reasoning agent.

A CommonKADS description of a ''task'' consists of two parts.

(1) The *task definition* is a declarative specification of the *goal* of the task. It describes *what* needs to be achieved. A CommonKADS task definition is equal to the notion of ''task'' in the Sisyphus glossary.
(2) The *task body* specifies a *procedure,* prescribing activities to accomplish the task. The task body describes *how* the goal can be achieved.

Roughly speaking, the task body can be seen as a method (PSM) for realizing the task (definition). However, in the CommonKADS vocabulary the term ''PSM'' is reserved for the generic (application-task independent) description of the method. In this view, the task body is strictly speaking the *instantiation* of a method for a

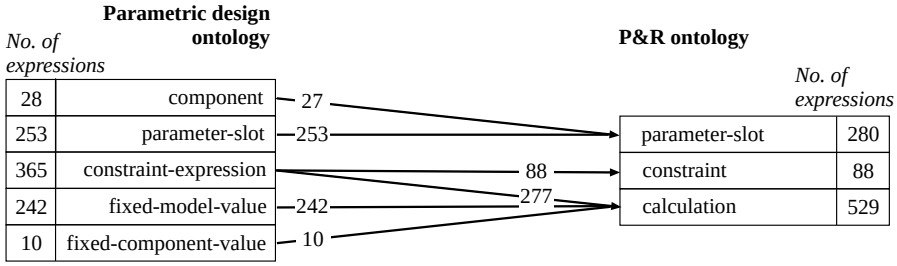


FIGURE 5. Overview of the transformation of expressions in the parametric-design ontology to the P&R ontology. The figures indicate the number of expressions involved. It can be seen that constraint expressions are split into two sub-sets. Only 88 constraint expressions are used as real 'constraints'. Components are transformed into parameters, if they have associated component models (this is true for all components but one). See Section 5 for more details about the transformation process.

particular task. Task body descriptions are usually annotated with the (generic) PSM that underlies it (see, for example, the CML description in Figure 7).

The task definition of the parametric-design task (shown in Figure 6) describes the overall goal of the task and its I/O. This task definition requires that the domain knowledge can be viewed in terms of a set of `parameters` representing the ''skeletal design'', and a set of `constraints` that involve these parameters.

The name ''propose-and-revise'' suggests that the first PSM that should be applied to this task is a decomposition of the top-level task into two sub-tasks: `propose` and `revise`. However, when examining the textual description of the method there appear to be at least three distinct sub-tasks at the first level of decomposition, namely:

TABLE 1
Three knowledge types representing additional ontological commitments required by the P&R method

Knowledge type	Domain model	Description
`fix`	`fixes`	Fix knowledge represented as a ternary relation between a `constraint`, an `operation`, and a `preference rating`. An `operation` is also a relation which has two possible sub-types: (i) a unary relation such as `upgrade` defined on a ''component-model'' parameter, or (ii) a binary relation such as `assign, decrease` and `increase` which holds between a parameter and a mathematical expression.
`order`	`parameter-order`	Representation of the heuristic that parameters should be assigned roughly in the order in which associated components are described in the VT data description.
`par-expression`	`initial-values`	Initial assignments to parameters, based on domain heuristics.

```
TASK parametric-design;
  TASK-DEFINITION
    GOAL:
      "find a design that meets the requirements and
       satisfies a set of constraints";
    INPUT:
      skeletal-design: "the set of system parameters";
      requirements: "requirements in the form of a set of parameter/value
        pairs";
    OUTPUT:
      design: "set of assigned parameters";
```

FIGURE 6. Task definition of the top-level task.

(1) `propose` a design extension;
(2) `verify` the current design;
(3) `revise` the design, if necessary.

This is in line with the decomposition provided by the ''PCM''-class of methods for routine design as described by Chandrasekaran (1990). In PCM-type methods the top-level design task is decomposed into four possible sub-tasks: propose, verify, critique and modify. The goal of the verification task is to check a proposed (partial) design. Critiquing is concerned with blame assignment in cases were verification indicates that there is a problem. As the verification of P&R produces a constraint violation which directly serves as input for the `revise` (i.e. modify) task, there is no explicit separation between verification and critiquing in this method. The direct link between verification and revision is possible, because P&R assumes that there is knowledge linking constraints directly to possible modifications (''fixes'', see Table 1).†

The task body in Figure 7 is the result of applying this method to the top-level task. A task body usually introduces new vocabulary to describe the relations between the sub-tasks. In the decomposition of the `parametric-design` task the following terms, or ''task roles'', are introduced.

(1) *Extended design:* this term is used to denote the set of parameters to which values have been assigned.
(2) *Design extension:* a new parameter assignment.
(3) *Violation*: a constraint (in the P&R sense) that was violated.

The control structure of the task body is stated in a form of pseudo-code. Arrows are used to distinguish input and output. The parametric-design task starts off with proposing a design extension (i.e. a new parameter value). The value is checked to see whether it introduces a constraint violation. If this is the case, the revise task is invoked with the violated constraint as input. This process is repeated until the `propose` task is not able to produce new design extensions. Parameters in the skeletal design have been assigned a value. If, for some reason, the `revise` task

† This statement is not true for SALT, the knowledge acquisition system supporting P&R. In SALT the acquisition of fix knowledge is guided by explaining to the user what are the possible contributors of a constraint violation. This is in fact a form of critiquing.

```
TASK-BODY
  TYPE: COMPOSITE;
  SUB-TASKS: init, propose, verify, revise;
  PSM: "PCM-type decomposition";
  ADDITIONAL-ROLES:
    extended-design: "current set of assigned parameters
      represented as a set of tuples <parameter, value>";
    design-extension: "proposed new element of the extended design";
    violation: "violated constraint";
  CONTROL-STRUCTURE:
    "parametric-design(skeletal-design + requirements -> design) =
       init(requirements -> extended-design)
      REPEAT
        propose(skeletal-design + extended-design -> design-extension)
        extended-design := design-extension JOIN extended-design
        verify(design-extension + extended-design -> violation)
        IF <some violation>
          THEN revise(extended-design + violation -> extended-design)
      UNTIL <the propose task fails to produce a design extension>
      design := extended-design";
END TASK parametric-design;
```

FIGURE 7. Task body of the parametric-design defining the first level of decomposition, as well as the control imposed on the decomposition.

fails, the overall design task fails. Note that there is no check in P&R whether all relevant parameters have been assigned a value. The assumption is that the knowledge underlying the propose step is capable of generating values for all relevant parameters. The main reason such an explicit completeness check is missing in P&R is probably the fact that there is no explicit component structure, that would allow the identification of relevant parameters.

The task body contains one additional task not mentioned in the PCM method. This `init` task can be viewed as an "administrative" task which ensures that the original input of the task is transformed into the form necessary for the sub-tasks. Here, the initialization task records the requirements as parameter values, thus providing the first version of the extended design.

The data flow between the sub-tasks is represented graphically in Figure 8. This

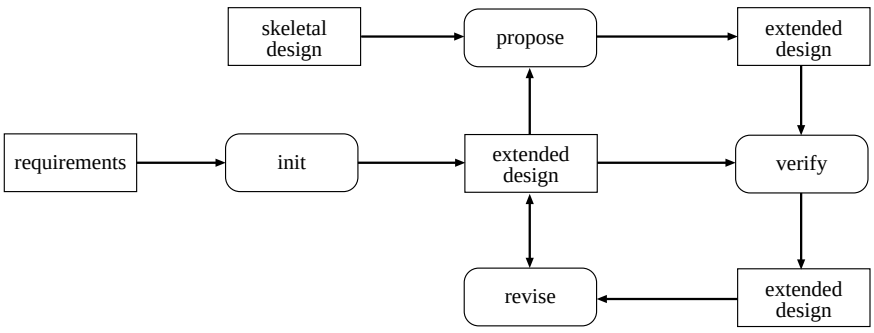


FIGURE 8. Top-level data flow in the parametric design task.

```
INFERENCE select-parameter;
  OPERATION-TYPE: "select";
  INPUT-ROLES:
    parameter-set -> SET(parameter);
    current-assignments -> "set of tuples <parameter, value>";
  OUTPUT-ROLES:
    selected-parameter -> parameter;
  STATIC-ROLES:
    order IN parameter-order;
    par-expression IN initial-values;
    calculation IN calculations:
  SPEC:
    "Select a parameter from the input set that has not been assigned a
    value and for which the preconditions that the domain knowledge poses
    for computing the value are fulfilled.

    The expressions in the domain model initial-values are evaluated as
    soon as possible.
    A heuristic ordering of components is used to rank the set of para-
    meters. (see VT description, start of Sec. 5)";
 END INFERENCE select-parameter;

 INFERENCE specify-value;
   OPERATION-TYPE: "compute";
   INPUT-ROLES:
     selected-parameter -> parameter;
     current-assignments -> "set of tuples <parameter, value>";
   OUTPUT-ROLES:
     parameter-assignment -> "single tuple <parameter, value>";
   STATIC-ROLES:
     par-expression IN initial-values;
     calculation IN calculations;
   SPEC:
    "Specify the value of a parameter by interpreting the formulae
    provided by the domain models, using the set of current assignments to
    retrieve already computed values.";
 END INFERENCE specify-value;
```

FIGURE 9. Inferences involved in the propose task.

figure should be interpreted as a *provisional* inference structure. It fulfills the role of a working hypothesis in the knowledge engineering process. It can (and will) be refined in the process of model construction, e.g. through task decomposition. For clarity, rounded boxes are used in the diagrams to refer to functions that can be decomposed further, in contrast to ovals which denote leaf functions (inferences in the CommonKADS vocabulary, see further).

4.2. PROPOSE

The method underlying the propose task is a strong form of decomposition. Invocation of this task does not produce a full design, but the smallest possible extension of an existing design, namely one new parameter assignment.

The input and output roles refer to the objects that are manipulated by the inference. The role label is supposed to be indicative of the way in which the object

is used *within* an inference, e.g. `selected parameter`. The specification of I/O roles also provides a pointer to the types of domain objects that can play this role, e.g. `parameter`. The static roles point to the domain knowledge that is used but not changed within the inference. A static role specification is of the form ⟨ type ⟩ *IN* ⟨ domain model ⟩, meaning that the inference uses expressions of the specified type from the domain model indicated. In the two sample inferences expressions from several domain models are used. The order in which parameters are selected is guided by heuristic knowledge as provided by the remark at the start of Section 5 of the VT data description:

> ''The subsections are ordered roughly in the order that values can be computed for design parameters.''

This is heuristic control knowledge (see `parameter-order` in Table 1). An alternative method would have been to analyse the constraint dependencies and derive from these an optimal ordering.

A precise description of the internal details of inferences is not assumed to be part of a CommonKADS expertise model; it is part of the system design activity, in which a suitable computational method is selected for the inference. Nevertheless, the knowledge engineer can informally describe the typical reasoning process in the ''spec'' slot. Also, an optimal ''operation-type'' can be specified. This operation type should be selected from the library of formally defined inference schema's developed by Aben (1995). Operation types can be used in several ways, e.g. as input for verification through formal specification, or as an index for computational method selection.

Inferences introduce their own role vocabulary, such as `current-assignments`. The invocation of an inference from a task indicates how task roles map onto inference roles. This distinction between task roles and inference roles allows the knowledge engineer to distinguish between, on the one hand, names that refer to the overall role of objects in problem solving (e.g. `skeletal-design`) and, on the other hand, names that refer to the role of an object within one inference step (e.g. `parameter-set`). Figure 10 shows the data dependencies between the inferences in the `propose` task. Such a figure is called in CommonKADS an ''inference structure''. The names in the boxes denote inference roles. The annotations represent the corresponding task role. In Figure 11 the full CML specification of the `propose` task is given. This task is defined as a *primitive* task, as it requires no further task decomposition: all sub-tasks are inferences.

4.3. VERIFY

The verification task in P&R applies a simple form of constraint evaluation. The method used here is to perform domain-specific calculations linked to the constraints. Two inferences are defined for this task (Figure 12). The `specify` inference produces the constraints that are associated with a parameter. As pointed out in Section 3, the term ''constraint'' as used in the method-oriented ontology of P&R refers to a sub-set of the constraint expressions in the domain. The `evaluate` inference uses a simple form of deduction to find out whether a constraint is consistent with the parameter assignments. The `verify` task invokes the two inferences defined above and returns the name of a constraint, if a violation is

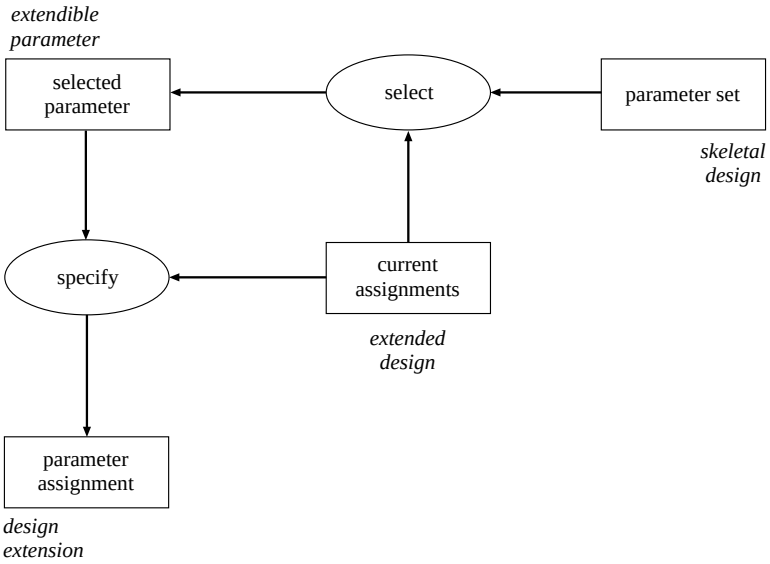


FIGURE 10. Inference structure of the `propose` task. Boxes denote inference roles; ovals denote inferences.

```
TASK propose;
  TASK-DEFINITION
    GOAL: "propose a design extension";
    INPUT:
      skeletal-design: "set of parameters to which values needs to be
                        assigned";
      extended-design: "set of tuples <parameter, value>";
  OUTPUT:
    design-extension: "new tuple <parameter, value>";
  TASK-BODY
    TYPE: PRIMITIVE;
    SUB-TASKS: select-parameter, specify-value;
    PSM: "single element decomposition";
    ADDITIONAL-ROLES:
      extendible-parameter: "parameter for which a value will be proposed";
    CONTROL-STRUCTURE:
      "propose(skeletal-design + extended-design -> design-extension) =
       select-parameter(skeletal-design + extended-design -> extendible-
       parameter)
       specify-value(extendible-parameter + extended-design -> design-
       extension)";
END TASK propose;
```

FIGURE 11. Specification of the `propose` task.

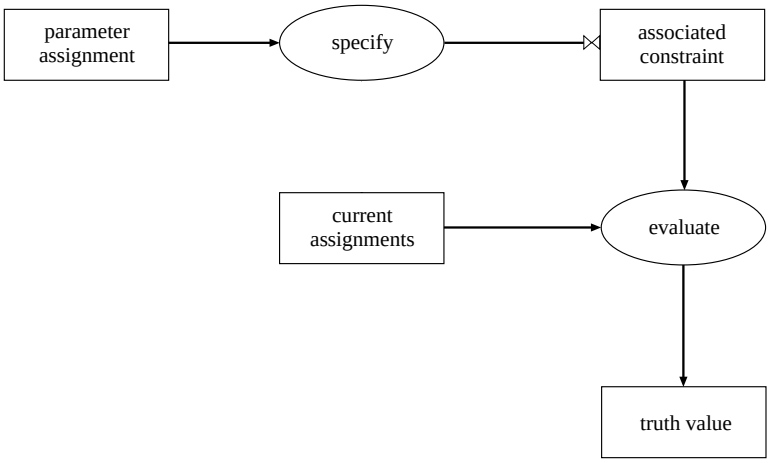


FIGURE 12. Inference structure of the `verify` task. The ⋈ symbol indicates a set. In this case it indicates that the output of `specify` is a set of constraints. Input to `evaluate` is one element of this set.

found. Verification is also a primitive task, as it invokes only inferences. In Figure 12 the data dependencies between the two inferences are shown in the corresponding inference structure.

4.4 REVISE

The `revise` task in P&R implements a quite specific strategy for modifying the current design, whenever some constraint violation occurs. To this end, the task requires a knowledge about fixes (see Table 1). Input to the task is a constraint violation. The goal of the `revise` task is to change the extended design in such a way that it is consistent with the violated constraint. This goal is realized by applying combinations of fix operations which change the value of parameters, and subsequently propagating these changes through the network formed by the computational dependencies between parameters. This leads to the following task decomposition of `revise`.

Find Fixes. Generate an ordered list of fix combinations that can be applied to the extended design to repair the violation.

Apply fix combinaiton. Create a temporary version of the extended design to which a fix is applied.

Update. Recompute the parameters which are computationally dependent on the parameters changed by the application of a fix.

Verify. Verify whether the new values of parameters changed by fixes or fix propagations are consistent with their constraints. This task is the same one as the `verify` task described previously.

The application of a fix may introduce new violations. Fixing these new violations would involve a recursive invocation of `revise`. P&R tries to reduce the complexity of `parametric design` by disallowing recursive fixes. Instead, if the application of a fix introduces a new constraint violation, the fix is discarded and a new combination of fixes is tried.

The method underlying the `revise` task avoids unnecessary fix propagation, by verifying the effects of afix application as soon as possible. We identified three situations in which verification should be carried out.

- After the application of a single fix in a fix combination, the constraints on the parameter which is changed by the fix should be verified. If such a constraint is violated, application of the rest of the fix combination should be aborted. During this step only those parameters should be updated that are computationally dependent on fixed parameters and also influence constraints on this parameter.
- After the application of a fix combination as a whole one should verify whether the constraint violation which triggered `revise` has been resolved. Up to this point only those parameters should be updated which influence the verification of the constraint.
- Only when the original constraint violation has been resolved, the other consequences of the application of the fix combination should be propogated. Finally all remaining parameters that were updated should be verified.

This leads to one additional sub-task for `revise`:

Find constraint influences. Find those parameters that directly or indirectly influence the verification of some constraint.

We have included the CML specification of the `revise` task in Figure 13. Six new inferences were specified for `revise` (see Table 2). The task also invokes inferences specified for `propose` and `verify`. We have omitted the specification of the control in sub-tasks involved in `revise`. This description is a very detailed specification of the algorithms employed by P&R, and it is debatable whether these detailed descriptions should be part of a knowledge-level model (see also the discussion section).

We limit the description here to some remarks about two issues that came up during the specification of `revise`.

Antagonistic constraints. The P&R problem-solving method limits the search space for parametric design by disallowing ''recursive fixing''. Whenever the application of a fix gives rise to a new constraint violation, the fix is rejected and no attempts are made to fix the new violation. A disadvantage of this search strategy is that it corrupts the declarative nature of the domain knowledge involved. The meaning of certain constraints is dependent on the order in which they are applied by the PSM. Such dependencies between constraints through fix ordering (termed ''antagonistic constraints'' by Marcus & McDermott, 1989) can be defined as follows.

Let *changed-by*(`constraint`) be the set parameters that might be changed by applying a fix for a violation of the constraint. The parameter might either be changed by a fix operation itself or by propagation of the fix.
Let *influences*(`constraint`) be the set of parameters that, directly or indirectly, influences the verification of a constraint.
Constraint A and B are antagonistic iff

$$\textit{changed-by}(A) \cap \textit{influences}(B) \neq \varnothing$$

Antagonistic constraints may prevent the system from finding a solution. For example when a violation of the constraint `hoist-cable-traction-ratio` is

```
TASK revise;
  TASK-DEFINITION
    GOAL:
      "modify the current design in such a way
      that no constraints are violated";
    INPUT:
      extended-design: "set of tuples <parameter, value>";
      violation: "constraint violated by the last extension";
    OUTPUT:
      revised-design: "revised extended design";
  TASK-BODY
    TYPE: COMPOSITE;
    SUB-TASKS: find-constraint-influences, find-fixes,
               apply-fix-combination, update-fix, eval-constraint, verify;
    PSM: "domain-specific revision strategies + dependency-directed
         backtracking";
    ADDITIONAL-ROLES:
      fix-combinations: "list of fix combinations";
      fix-combination: "sequence of actions which might resolve a violation";
      constraint-influences: "those parameters which are directly or
              indirectly influence the verification of the violated
  constraint";
      affected-parameters: "parameters affected by a fix combination";
    CONTROL-STRUCTURE:
      "revise(extended-design + violation -> revised-design) =
         find-fixes(violation -> fix-combinations)
         REPEAT
           1. Select a fix combination
           2. Apply the fix combination to the affected-parameters
           3. If step (2) did not introduce any violations then:
                 (3a) find the constraint-influences
                 (3b) update the parameters belonging to the intersection
                      of constraint-influences and affected-parameters
                 (3c) evaluate the original constraint
                 (3d) if the violation has been resolved then
                      update the rest of the parameters in affected-
                      parameters
                 (3e) verify each update resulting from step (3d)
       UNTIL revised-design is consistent";
END TASK revise;
```

FIGURE 13. Specification of the `revise` task.

TABLE 2
Overview of inferences involved in the `revise` *task*

Inference	Description
`specify-fixes`	Find all fixes for a violation
`order-fixes`	Construct and sort all possible combinations of fixes
`apply-fix`	Recompute the value of a parameter
`parameter-children`	Find all parameters that are dependent on a parameter
`parameter-parents`	Find all parameters that directly or indirectly determine the value of the input parameter
`constraint-parameters`	Find parameters used in a constraint

repaired before a violation of the constraint `machine-beam-section-modulus` P&R fails to find a solution. However, when the repair order is reversed a solution is found. The KA front end to VT (SALT, Mardus & McDermott, 1989), tries to deal with these antagonistic constraints by eliciting heuristic orderings of constraint fixes. An alternative is to derive these constraint dependencies at compile time, and to use the direction of the antagonistic relation to establish the order in which parameters should be derived.

Internal dependencies in a fix combination. The current conceptualization of P&R does not take into consideration that fixes in a fix combination may be dependent of each other. These dependencies can be of two types.

(1) Fixes may use parameters in their computation, which are changed by another fix in the combination.
(2) Fixed parameters may have constraints which are (in)directly related to parameters which are changed by another fix.

These observations have two consequences. Firstly, fixes in a fix combination should be ordered in such a way that dependent fixes are applied after fixes that depend on it. Secondly, these dependencies should be propagated before the application of a dependent fix.

4.5. SUMMARY OF PROBLEM-SOLVING METHODS IN P&R

P&R is in fact a conglomerate of related methods. We have tried to indicate what the types of methods are that appear in the decomposition of the parametric-design task as dictated by P&R. For this purpose we used the task analysis framework defined by Chandrasekaran (1990). Summarizing, the task decomposition as described in this section encompassed the following method.

- The decomposition of the top-level task `design` into three sub-tasks, `propose`, `verify` and `revise` is an instance of the class of PCM-class of design methods.
- The `propose` task is realized through a particular type of decomposition method, namely a design plan consisting of subsequent steps of single parameter assignments.
- The method used by `verify` method employs domain-specific calculations provided by the expressions attached to the constraints (e.g. the calculation of the traction ratio).
- The `revise` task uses domain-specific revision strategies provided by the fix knowledge. The control is governed by the dependency-directed backtracking method.

Figure 14 shows the relation between tasks and methods graphically.

5. Design and implementation

In CommonKADS expertise modelling is followed by an explicit design step resulting in a design model.† A CommonKADS design model (Van de Velde,

† Other inputs for the design model may come from the task, agent and communication models, but those models are not covered in this article.

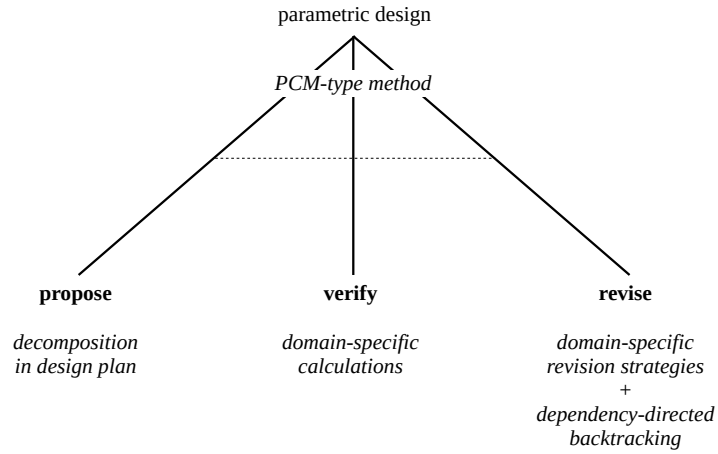


FIGURE 14. Task decomposition generated by P&R. The italic annotations characterize the methods on which the decomposition is based.

Duursma, Schreiber, Terpstra, Schrooten, Golfinopoulos, Olsson, Sundin & Gustavsson, 1994) consists of an *architecture design,* an *application design,* and a *platform design.* The architecture design defines an abstract computational engine that contains the computational primitives for realizing the application (e.g. representation languages, implemented algorithms). Application design describes how the ingredients of the expertise model (and/or task, communication and agent models) are mapped onto the architecture. The platform design describes the hardware and software platforms on which the application is implemented.

One important principle that we apply during design is the notion of *structure-preserving* design: both the content and the structure of the information in the expertise model should be preserved as much as possible in the final system. This ensures the explainability (in knowledge-level terms) and maintainability of the system (Schreiber, Wielinga & Breuker, 1993: chapter 6). Structure-preserving-design has a parallel in other approaches, e.g. the program writer in EES (Neches, Swartout & Moore, 1985). The first step in structure-preserving design is to define CommonKADS specific constructs on top of the architecture as defined in the architecture design. This CommonKADS specific viewpoint on the architecture then provides a framework for enumerating and documenting application-specific design decisions (Terpstra & Schrooten, 1993).

5.1. ARCHITECTURE AND PLATFORM DESIGN

For realizing the VT application we used the SIADL† implementation of a CommonKADS-specific architecture (Terpstra, 1994). This architecture implementation was based on the previous architecture used in Sisyphus-I (Schreiber, 1994) with additional facilities for handling multi-level ontologies and ontology transformations. Using such an environment the knowledge engineer can concentrate on the application design activity. The environment also enables the transformation of

† Simulated Application Design Language.

CML descriptions as (skeletal/partial) application design specifications. The SIADL system was developed in SWI-PROLOG, a public-domain Prolog that is available on both Unix and Windows platforms.

5.2. APPLICATION-DESIGN: DOMAIN KNOWLEDGE

To support structure-preserving design, the SIADL environment provides a default domain-knowledge representation. This representation is a simple tuple-oriented representation, and acts as an intermediate between the CML and representations defined for other specification languages (e.g. KIF) or representations which are optimized towards a computational technique (''symbol level representations''). As the language has a formal syntax for domain axioms, it can be used for the realization of mappings through transformation of domain expressions.

The SIADL environment provides facilities for defining mappings. Two types of domain-model mappings are being distinguished.

Ontology mapping. An ontology mapping maps a set of expressions onto a set of expressions that has a different ontology. This means that semantics of the expressions is changed. The mapping between the parametric-design domain model and the P&R domain models is example of an ontology mapping. A P&R specific meaning is attached to the transformed expressions. For example, the '' = '' symbol is interpreted in calculation formulae as an assignment operator. The semantics of the expressions used by P&R is thus closely connected to their role in problem solving (in this case, computing the value of the variable at the left-hand side of the equation).

Representation mapping. A representation mapping maps expressions in one representation onto expressions in another representation under the assumption that the semantics of the expressions is preserved. This implies that the representations share an ontology. This type of mapping is similar to the translations provided by the ONTOLINGUA software.

The mapping from the VT domain theory onto expressions in the parametric-design domain model is largely a representation mapping: the CML ontology only differs marginally from the Ontolingua one. Only with respect to the additional distinction made between `component` and `component-model` the, transformation is of the ontology-mapping type.

The VT implementation that we constructed is an example of how multi-level ontologies and the different types of mappings enable knowledge reuse. Figure 15 shows how the ONTOLINGUA knowledge base for VT was made available in our SIADL environment. This process consisted of the following steps.

Step 1: Representation mapping Ontolingua $\mapsto$ KIF. The starting point is the VT domain theory in the ONTOLINGUAform (the file *vt-domain.lisp*). This theory is translated into the KIF form by the Ontolingua-to-KIFtranslator. KIF was chosen as an intermediate representation, because the KIF format is closest to Prolog.

Step 2: Representation mapping KIF $\mapsto$ Prolog. The file *vt-domain.kif* is parsed into a Prolog-readable file *vt-domain.pl.* The transformations are mainly lexical/syntactical, e.g. handling the different lexical conventions for symbols/atoms. The parser is *not* a full KIF-to-Prolog parser: it handles only the sub-set of KIF used in the VT application.

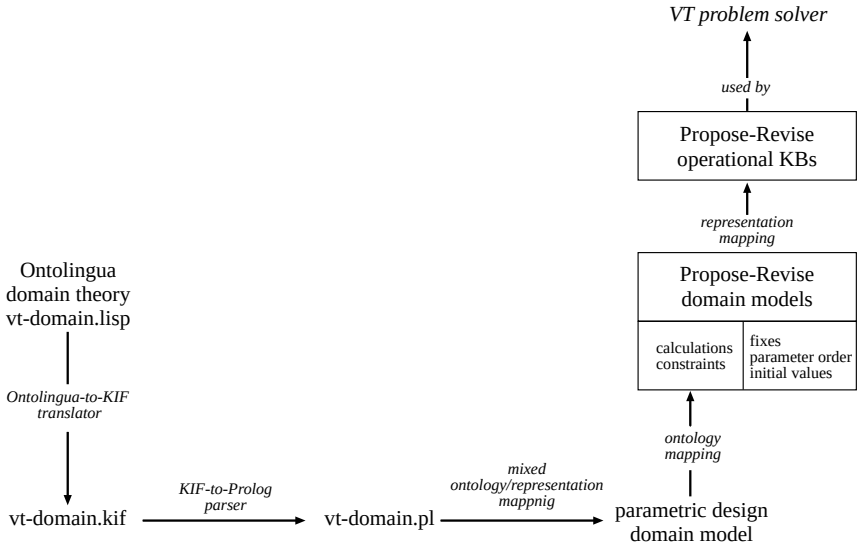


FIGURE 15. Mapping process from the VT domain theory in Ontolingua/KIF to the CommonKADS domain models based on the parametric design and P&R ontologies.

Step 3: Mixed mapping vt-domain $\mapsto$ parametric-design. The file *vt-domain.pl* is subsequently interpreted by a set of mapping rules that rewrite the domain expressions into statements of the types defined in the parametric-design ontology. These mapping rules handle the differences between the ONTOLINGUA and the CML ontology definitions (cf. the discussion in Section 3.1). The result is the `parametric-design` domain model, represented in the default representation of our environment.

Step 4: Ontology mapping parametric-design $\mapsto$ P&R. The task-type oriented ontology in turn is mapped through a second set of rewrite rules onto those constructs in the (method-oriented) P&R ontology that specify viewpoints on the parametric-design task ontology (see the mapping rules in Figure 4). This mapping creates a set of domain models in the P&R format (see also Figure 5).

In addition, this method-oriented ontology defines a number of conceptualizations that are specific for this method, and thus have no counterpart in the task-type oriented ontology. An example in the VT domain is the fix knowledge. These were added manually to the appropriate method-oriented domain models `fixes`, `initial-values` and `parameter-order`.

Step 5: Representation mapping P&R $\mapsto$ computational technique. The VT application uses computational techniques that may require a specific representation optimized towards efficient computation (see the next sub-section). Therefore, an additional mapping may be needed that transforms certain domain models into the symbol-level representation imposed by a computational technique. An example is the representation required by the formula evaluator employed in our application.

Figure 16 shows three examples of mapping rules used for the transformation of the parametric-design domain model into the P&R representation (Step 4). The first two rules specify the mapping from both `parameter-slot` and `component` to

```
mapping(parameter,                      % name of construct in destination
                                          ontology
        parameter(P),                   % representation in destination
                                          ontology
        component(P),                   % matching representation in the
                                          source ontology
        model_ref(has_model(P, _))).    % conditions/actions on source
                                          ontology construct
mapping(parameter,
        parameter(P),
        parameter_slot(P)).
mapping(calculation,
        calculation(Expr, Output, Inputs),
        constraint_expression(Expr),
        (ontology_ref(mathematical_expression, Expr)
        , rewrite(Expr, Output, Inputs))).
```

FIGURE 16. Three examples of mapping rules for the transformation of the parametric-design domain model into the P&R representation. The first rules specify the mapping from `parameter-slot` and `component` to `parameter`. The third rule is one of the mapping rules that generates tuples of the `calculation` relation. See the text for explanation of the structure of the mapping rules.

`parameter`. The third rule is one of the mapping rules that generates tuples of the `calculation` relation. Each mapping rule has four arguments. The first argument indicates the construct in the destination ontology that the mapping rule generates. The second argument specifies the representation of this construct in the destination domain model. The third argument specifies the representation of expressions in the source domain ontology to which the mapping rule applies. The last argument is optional, and specifies conditions and/or actions on the source expressions. The conditions can be type checks (`ontology-ref`) or references to the existence of other source expressions (`model-ref`). The actions are typically rewrite operations, such as the retrieval of the variables involved in calculations (see the third mapping rule). The underlying transformation techniques make use of Prolog unification.

Table 3 lists the number of mapping rules used in steps 3–5, plus the number of ontology constructs involved in these rules.

5.3. APPLICATION-DESIGN: INFERENCE KNOWLEDGE

In application design *inference procedures* need to be specified that define the computational realization of inferences specified in the expertise model. An inference procedure consist of a set of domain-knowledge retrieval operations and calls to computational methods. The relation between inferences and inference procedures is not necessarily one-to-one. The knowledge engineer may decide to organize the reasoning process differently for efficiency reasons. Typically, there is a trade-off between complete structure-preserving design and designing a very efficient system. One of the purposes of the CommonKADS application design activity is to document such trade-off decisions explicitly.

In the VT case three design decisions were taken to improve the efficiency of the `propose` task.

(1) In the specification of the inference `select-parameter` it was indicated that a

TABLE 3
Overview of transformation steps 3–5

Source	Destination	Type	No. of constructs	No. of rules
vt-domain (Prolog)	↦parametric-design	Mixed	15	15
parametric-design	↦P&R	Ontology	3	7
P&R	↦formula-evaluator	Representation	6	6

heuristic ordering was to be imposed on the parameter set (the skeletal design). Computing this ordering each time the inference is invoked would, however, be very inefficient. For this reason, a design decision was made to represent the skeletal design as an ordered list of parameters, so that a simple `select-first` method could be used by the inference procedure implementing `select-parameter`. To achieve this, an inference procedure `order-parameter-set` was added to the initialization task. This procedure accesses the domain knowledge provided by the static role `parameter-order`.

(2) The specification of `select-parameter` mentions that formula's specifying initial values should be evaluated first. Also, both the `select-parameter` and the `specify-value` inference are reused in the `revise` task, but using only the domain model `calculations`. For these reasons, both the select and the specify step were specialized into two sub-types, each accessing a separate domain model (see inference procedures 3–6 in Table 4).

(3) In the expertise-model specification both `select-parameter` and `specify-value` retrieve the formula for assigning a value. This is inefficient, as only the first retrieval is necessary. This formula can be passed on to the procedure implementing `specify-value` as an additional input argument.

This type of small modification is typical for the design process in CommonKADS. The additional input is not necessary from an knowledge-level point of view: the system would produce exactly the same result without it. It would only take more time.

Table 4 lists the set of inference procedures used in the VT implementation. For each inference procedure, the associated computational methods are shown. Most procedures invoke one or two simple general-purpose methods (e.g. `predicate-sort, transitive-closure`) or just retrieve domain knowledge through role instantiation. The only exception is the inference procedure `order-fixes`, which uses a method specific for P&R.

5.4. APPLICATION-DESIGN: TASK KNOWLEDGE

The design decisions with respect to task knowledge are usually not very complicated. The pseudo-code used in the task descriptions has to be transformed into *task procedures.* In cases where the set of inference procedures is different from the inferences, the corresponding task procedure has to be updated accordingly. For example, the task procedure for the `propose` task has to take into account that the two inference procedures involving initial values are applied before the other two.

Other detailed specifications that may have to be added mainly concern

TABLE 4

Computational methods used to realize inference procedures (*i.e., the design equivalent of an inference*). *Almost all computational methods are simple standard algorithm. One inference procedure uses a P&R specific algorithm* (`order-fixes`)

Inference procedures	Procedural description	Methods
`order-parameter-set`	Sort parameter set based on heuristics	`predicate-sort`
`init-assignments`	Record the requirements as initial parameter assignments	`retrieval`
`select-initial-parameter`	Select the first parameter for which an initial value exists	`select-first` `retrieval`
`specify-initial-value`	Assign value to parameter	`unification`

`select-calculation-parameter`	Select the first parameter for which a calculation exists in which all input variables have a value	`select-first`
		`retrieval`
		`set-completeness`

`specify-calculation-value`	Evaluate a formula given the values of parameters in the current set of assignments	`formula-evaluator`
`specify-constraints`	Find all constraints on a parameter	`retrieval`
`evaluate-constraint`	Evaluate a constraint, given the current set of assignments	`formula-evaluator`
`specify-fixes`	Find all fixes for a violation	`retrieval`
`order-fixes`	Construct all possible combinations of fixes	`power-set`
	Sort the fix combinations	`predicate-sort`
	Compute update dependencies between fixes	P&R-specific
`apply-fix`	Recompute the value of a parameter	`formula-evaluator`
`parameter-children`	Find all parameters that are computationally dependent on input parameter	`transitive-closure`
`parameter-parents`	Find all parameters that directly or indirectly determine the value of the input parameter	`transitive-closure`
`constraint-parameters`	Retrieve all parameters used in a constraint	`retrieval`

bookkeeping activities (storing immediate results). Task procedures are also the place where actions specified in the *communication model* are integrated: I/O activities such as reading the requirements, printing trace information, etc. As VT is modelled here largely as a ''batch'' system, this was not an important issue in this case.

5.5. SAMPLE TRACE

This section lists some fragments of the trace that was generated for the sample problem in the VT documentation. The trace information displayed by the system is just meant to show the essentials of the reasoning process in terms of the expertise model. No effort was made to build any sort of nice interface or complete trace facility.

```
Starting VT
User datum: car_cab_height = 96
User datum: car_capacity_range = 3000
. . .
. . .
User datum: platform_width = 70

Extension sling_model = sling_model_m01
Extension motor_generator_model = motor_generator_model_m03
Extension machine_model = machine_model_m01
Extension machine_groove_model = machine_groove_model_m02
. . .
. . .
. . .
Extension safety_beam_model = safety_beam_model_m01
Extension safety_beam_load_maximum = 8000
Extension safety_beam_height = 9
Extension safety_beam_constant = 2.250000
```

FIGURE 17. Trace of some user specifications and design extensions.

The first fragment (see Figure 17) shows parts of the trace of the *init* task and the *propose* task. No violations were found for the design extensions generated in this fragment.

Figure 18 shows a fragment in which a constraint is violated and needs to be fixed. The constraint involved is `hoist-cable-traction-ratio`. Fifteen fix combinations were found (for reasons of space only a few are shown in Figure 18). The eighth combination was successful. As most fix operations (e.g. upgrade, increase, decrease) within a fix combination can be repeated several times, one fix combination actually defines a space of possible fixes. Propagating the fixes resulted in updating 64 parameters of the extended design.

The system, running on a Sun-Sparc 10, solved the problem in 271 seconds (average speed 27K Lips).

6. Discussion

The goal of this paper was to provide a good data point for comparison of knowledge modelling approaches, with respect to the description of the ontology and the problem-solving model. We discuss each of these separately, and also briefly address the issues with respect to the system design process and the relation with Sisyphus-I.

(1) *Multi-level ontologies as landmarks for reuse.*

We have found that the use of different ontologies at different levels of generality is a powerful and indispensable tool for knowledge sharing and reuse. The description in Section 3 and Section 5 give evidence of this. The Ontolingua knowledge base, built at another site, on a different platform, using a different representation, could be used in our environment to access knowledge types needed by the application. This would not have been possible if the ontology of our application had just been phrased in pure method-specific terms. In the VT case approximately ninety percent of the total amount of knowledge required for the application could be

```
  Extension hoist_cable_traction_ratio = 1.853457 is violated by c_48_2:

machine_groove_model = machine_groove_model_m02 =>
hoist_cable_traction_ratio =<0.006555 * machine_angle_of_contact +
  0.755000

Found 15 fix combinations.

Trying fix combination:
fix: counterweight_to_platform_rear is counterweight_to_platform_rear −
   0.5 Cost : d3

Trying fix combination:
fix: compensation_cable_model is upgrade(compensation_cable_model)
  Cost : d6
fix: car_supplement_weight is car_supplement_weight + 100 Cost : d4

Trying fix combination:
fix: compensation_cable_model is upgrade(compensation_cable_model)
  Cost : d6
fix: car_supplement_weight is car_supplement_weight + 100 Cost : d4
fix: counterweight_to_platform_rear is counterweight_to_platform_rear −
  0.5 Cost : d3

Applied successful fix
Fix results
  compensation_cable_model: compensation_cable_model_m07 -->
    compensation_cable_model_m03
  car_supplement_weight: 0 --> 500
  counterweight_to_platform_rear: 5.250000 --> 1.750000
Propagated:
  counterweight_to_hoistway_rear: 6 --> 9.500000
  machine_angle_of_contact: 152.405213 --> 155.761553
  machine_sheave_to_deflector_sheave_diagonal: 54.886958 --> 53.071444
  machine_sheave_to_deflector_sheave_horizontal: 29.750000 -->
    26.250000
  car_cable_hitch_to_counterweight_cable_hitch: 54.750000 -->
    51.250000
...
...
...
  cable_load_unbalanced: 78.914250 --> 6.014250
  compensation_cable_length: 0 --> 993
```

FIGURE 18. A trace fragment of fixing the violation of constraint c_48_2.

reused through the structured mapping process. It actually meant a reduction of a number of weeks with respect to the effort of building the application.†

In this article a distinction was made between a general ontology for a task such as parametric design on the one hand and a more application-specific method-oriented ontology on the other hand. Although this distinction proved to be useful for this application, the borderline is not as firm as it looks, and it is also not the only possible borderline. For example, one could argue that a fix is a general notion that

† Even taken into account that various bugs were found and repaired in the VT domain theory.

can be found in any design task. One could also argue that the part-of structures of components are not task-type oriented, but represent in fact a more general, task-independent, notion. With respect to method-oriented ontologies it is likely that classes of methods share a number of ontological commitments. It is thus probably best to view the distinction between task-oriented and method-oriented ontologies as a *hypothesis* about one useful landmark in a spectrum of types of ontological commitments (Wielinga *et al.,* 1993). Such landmarks can guide the reusability and/or shareability of parts of a knowledge base.

The ontologies described in Section 3 should be seen as a first start. We make no claim, for example, that the parametric-design is valid to its full extent. We view the process of building reusable ontologies basically as an empirical process. Ontologies such as the ones proposed for VT have to be used in practice, and should then be refined modified, split into sub-theories, etc. according to the experiences gained. In our view, such an empirical process is the only way we can arrive at a useful set of reusable ontologies. The VT exercise is just a first step in this direction. To support this ontology-building process techniques for describing semantical mappings between ontologies and storing ontologies in a library with some ''reusability index'' will need to be developed. One approach to ontology indexing is proposed by van Heijst, Falasconi, Abu-Hanna, Schreiber and Stefanelli (1995).

(2) *PSM descriptions.*

The description of the problem-solving model is a fair example of how CommonKADS task and inference descriptions are constructed. In CommonKADS a PSM is a meta-level notion describing the rationale underlying a task/inference decomposition. The method itself is mentioned, but not explicitly represented in the description. The methods are considered to be part of the background knowledge of the knowledge engineer. We started looking for a KADS/CommonKADS description of the method, but none were available at that time. As we decided to model propose-and-revise as closely as possible, we could not reuse directly the method used for the office assignment problem. Therefore, we ended up constructing the CommonKADS description of P&R more or less from scratch, using Chandrasekaran's (1990) task analysis framework for design as a guideline. We did not keep records of this development process, but the effort spent was in the order of a few weeks. Some details of the `revise` task only became clear after the operationalization of the model in a running system.

In the mean time, the CommonKADS library book (Breuker & Van de Velde, 1994) has become available, so it is worthwhile to look *a posteriori* whether this would have been of help. The description of propose-and-revise in chapter 11 of this book would have provided the following structure.

- A decomposition of `propose` into three sub-tasks each of which can be found in our VT model: `sequence` (ordering of the parameter set), `select-unit` (select parameter), and `propose-assignment` (specify value).
- The `revise` task is decomposed into an `evaluate` task (corresponds to the VT `verify` task), and a `modify` task. For this last task five methods are mentioned, including `generic-fixes`. No further decompositions are provided.

It seems fair to conclude that the library would have given a first top-level description of P&R, but more or less on the same level as Chandrasekaran's

framework. It would not have provided the detailed descriptions presented in Section 4. For detailed PSMs such as the method used in the `revise` task, it would be worthwhile to consider how we can generate useful knowledge-level descriptions of the algorithms involved. A description of this part cannot be derived from the textual descriptions of this method. Providing support for this step through reusable models of optimization algorithms would have been extremely useful here. Researchers involved in Sisyphus are in fact already proposing such models (see for example the contributions to KAW'95 on this topic).

(3) *Rationale of a separate system-design activity.*

KADS has been critized for the fact that the knowledge-level analysis does not directly produce an operational system. We strongly believe, however, that a separate design activity is necessary for any realistic application. Operational knowledge-level models require that the operational interpretation of a knowledge-level description is fixed in advance. This is in general too strong a constraint, either leading to unacceptably inefficient systems, or to severely limited expressiveness of the modelling language. The small adaptations made during operationalization of the inferences involved in the `propose` task are typical design activities that increase the efficiency of the resulting system drastically. The first CommonKADS system, built by Magni (1993), was not optimized during design and solved the problem in approximately two hours. The few optimizations that were added for the application described in this article reduced the time required to solve the problem to about 4% of the original figure.

It would definitely be very undesirable to change the knowledge-level model each time such an efficiency decision is made. That would be cumbersome from a management point of view, but would also lead to a level of detail in the description that destroys its knowledge-level character.

(4) *Relation to Sisyphus-I*: *office assignment.*

A comparison between the model presented in this paper and the office-assignment model is difficult. Although we could have decided to solve the office-assignment problem with a method similar to P&R, we decided at the start of Sisyphus-I to model the protocol (featuring Sigi D.) as closely as possible (Schreiber, 1994). The result was a model for the `propose` task of an *allocation* problem. An allocation problem has characteristics that make it different from the type of design described in this paper. One prime difference is the fact that in allocation the domain consist of two groups of objects (consumers and providers). Mapping this on a P&R type problem would mean reformulating consumers and providers as parameters and values. This is in principle, possible, but it corrupts the neat distinction between the two groups. There are a number of commonalities between the two models with respect to the `propose` task. These commonalities are described in the CommonKADS library (Breuker & Van de Velde, 1994; Ch. 11), where the Sisyohus-I descriptions were used to describe possible decompositions of the propose task of P&R.

Paolo Magni and Marc van Velzen contributed to early versions of the VT model and systems. Anjo Anjewierden and Manfred Aben built parsers for CML, thus helping us to correct many small errors. We have profited from discussions with Gertjan van Heijst, Bob Wielinga, and other colleagues at SWI. We also wish to thank the participants of the Sisyphus meetings during KAW'94 and EKAW'94. We are grateful for the many things we learned from this collaborative effort of the knowledge engineering community.

The research reported here was carried out in the course of the KADS-II and the KACTUS projects. The KADS-II project was partially funded by the ESPRIT Programme of the Commission of the European Communities as project number 5248. The partners in this project were: Cap Gemini Innovation (France), Cap Programator (Sweden), Netherlands Energy Research Foundation ECN (The Netherlands), Eritel SA (Spain), IBM France, Lloyd's Register (United Kingdom), Swedish Institute of Computer Science (Sweden), Siemens AG (Germany), Touche Rosse MC (United Kingdom), University of Amsterdam (The Netherlands) and Free University of Brussels (Belgium). The KACTUS project is partially funded by the ESPRIT Programme of the Commission of the European Communities as project number 8145. The partners in this project are: Cap Gemini Innovation (France), LABEIN (Spain), Lloyd's Register (United Kingdom), Statoil (Norway), CAP Programator (Sweden), University of Amsterdam (The Netherlands), University of Karlsrule (Germany), IBERDROLA (Spain), DELOS (Italy), FINCANTIERI (Italy) and SINTEF (Norway). This article expresses the opinions of the authors and not necessarily those of the consortium.

References

Aben, M. (1995). *Formal methods in knowledge engineering.* Ph.D. thesis, University of Amsterdam, Faculty of Psychology. ISBN 90-5470-028-9.

Barthèlemy, S., Frot, P. & Simonin, N. (1988). *Analysis document experiment F4.* ESPRIT Project P1098, Deliverable E4.1, Cap Sogeti Innovation.

Breuker, J. A. & Van de Velde, W., Eds (1994). *The CommonKADS Library for Expertise Modelling.* Amsterdam: IOS Press.

Chandrasekaran, B. (1988). Generic tasks as building blocks for knowledge-based systems: the diagnosis and routine design examples. *The Knowledge Engineering Review,* **3,** 183–210.

Chandrasekaran, B. (1990). Design problem solving: a task analysis. *AI Magazine,* **11,** 59–71.

Magni, P. (1993). *Un ambiente per l'acquisizione della conoscenza.* Master's thesis, Universita degli Studi di Pavia, Italy. (In Italian).

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

Marcus, S., Stout, J. & McDermott, J. (1988). VT: an expert elevator designer that uses knowledge-based backtracking. *AI Magazine,* Spring, 95–111.

Neches, R., Swartout, W. R. & Moore, J. D. (1985). Enhanced maintenance and explanation of expert systems through explicit models of their development. *IEEE Transactions in Software Engineering,* **11,** 1337–1351.

Runkel, J. T., Birmingham, W. P. & Balkany, A. (1996). Solving VT by reuse. *International Journal of Human-Computer Studies,* **44,** 403–433.

Schreiber, A. T. (1994). Applying KADS to the office assignment domain. *International Journal of Human-Computer Studies,* **40,** 349–377.

Schreiber, A. T., Wielinga, B. J., Akkermans, J. M., Van de Velde, W. & Anjewierden, A. (1994a). CML: the CommonKADS conceptual modelling language. In L. Steels, A. T. Schreiber & W. Van de Velde, Eds. *A Future for Knowledge Acquisition. Proceedings of the 8th European Knowledge Acquisition Workshop EKAW'94, volume 867 of Lecture Notes in Artificial Intelligence,* pp. 1–25. Berlin: Springer-Verlag.

Schreiber, A. T., Wielinga, B. J. & Breuker, J. A., Eds (1993). *KADS: a Principled Approach to Knowledge-Based System Development, volume 11 of Knowledge-Based Systems Book Series.* London: Academic Press.

Schreiber, A. T., Wielinga, B. J., de Hoog, R., Akkermans, J. M. & Van de Velde, W. (1994*b*). CommonKADS: a comprehensive methodology for KBS development. *IEEE Expert,* **9,** 28–37.

TANK, W. (1992). *Modellierung von Expertise über Konfigurierungsaufgaben.* Sankt Augustin, Germany: Infix.

TERPSTRA, P. (1994). *An environment for application design.* ESPRIT Project 5248, Deliverable DM7.5a KADS-II/M7/UvA/072/1.0, University of Amsterdam, The Netherlands.

TERPSTRA, P. & SCHROOTEN, R. (1993). *CommonKADS specific design decisions and their notation.* Deliverable DM7.2b, ESPRIT Project P5248 KADS-II/M7.2/DD/UvA/043/1.1, University of Amsterdam, The Netherlands and Free University of Brussels, Belgium.

VAN DE VELDE, W., DUURSMA, C., SCHREIBER, G., TERPSTRA, P., SCHROOTEN, R., GOLFINOPOULOS, V., OLSSON, O., SUNDIN, U. & GUSTAVSSON, M. (1994). *Design model and process.* Deliverable DM7.1, ESPRIT Project P5248 KADS-II/M7/VUB/RR/064/2.1, Free University Brussels, University of Amsterdam, The Netherlands, Swedish Institute for Computer Science, Sweden and Cap Programmator, Brussels, Belgium.

VAN HARMELEN, F. & BALDER, J. R. (1992). $(ML)^2$: a formal language for KADS models of expertise. *Knowledge Acquisition,* **4**.

VAN HEUST, G., FALASCONI, S., ABU-HANNA, A., SCHREIBER, A. T. & STEFANELLI, M. (1995). A case study in ontology library construction. *Artificial Intelligence in Medicine,* **7,** 227–255.

WIELINGA, B. J., VAN DE VELDE, W., SCHREIBER, A. T. & AKKERMANS, J. M. (1993). Towards a unification of knowledge modelling approaches. In J.-M. DAVID, J.-P. KRIVINE & R. SIMMONS, Eds. *Second Generation Expert Systems,* pp. 299–335. Berlin: Springer-Verlag.