Int. J. Human–Computer Studies (1994) **40**, 349–377

Applying KADS to the office assignment domain

A. Th. SCHREIBER

University of Amsterdam, Social Science Informatics, Roetersstraat 15, NL-1018 WB Amsterdam, The Netherlands

In this article the KADS approach is used to model and implement the office assignment problem. We discuss both the final products (the model of expertise and the design) and the process that led to these products. Emphasis is put on modelling the problem in such a way that it closely corresponds to the behaviour of the expert in the sample protocol. The last section of the paper addresses the evaluation points raised by the initiators of Sisyphus.

1. Introduction

This article describes an exercise to model and implement the sample problem of the Sisyphus-91/92 project using the KADS approach. KADS has been and is being developed in a series of ESPRIT projects (P12, P1098, P5248). The most well-known ingredient of KADS is the so-called "four-layer model": a conceptual framework for modelling the problem-solving expertise in a particular domain using several layers of abstraction. The layers allow the knowledge engineer to describe the problem-solving process in a domain-independent fashion. Another central aspect of KADS is the distinction between (i) a conceptual model of expertise independent of a particular implementation, and (ii) a design model specifying how a model of expertise is operationalized using particular computational and representational techniques.

The KADS approach has evolved over the years. Here, we apply KADS as it is described in Wielinga, Schreiber and Breuker (1992) and Schreiber, Wielinga and Breuker (1993). We do not take recent developments in the KADS-II project into account.

This article is organized as follows. In Section 2 a brief account is given of the steps that were taken to arrive at the model for the office-assignment task-domain. Section 3 discusses some initial observations that came out of a first global analysis of the problem description. Sections 4–6 describe the model of expertise that was constructed for this application. Section 7 discusses the step from model of expertise to design and implementation. Finally, in Section 8 the proposed solution to the problem is evaluated with respect to the questions raised in the problem description.

2. Modelling the office assignment problem

The problem description basically consists of two parts:

1. A description of the major entities (employees, rooms, projects) and relationships (hierarchies, project assignments, floor plan) in the sample domain.
2. A think-aloud protocol showing how an expert solves a particular office assignment problem.

1071-5819/94/020349 + 29$08.00/0

As there is only one protocol, it can occur in the remainder of this article that there is not sufficient information to make a particular (modelling) choice. Such a situation usually gives rise to a *knowledge engineering* (KE) *goal*: a topic for which further knowledge elicitation and/or analysis is necessary. We will point to these KE goals in the text and state what kind of assumptions we have made about such goals' outcomes in building the model.

We should also mention here that it is our goal to build a model and a system that reflects as closely as possible the reasoning process of the expert. It is not our goal to find an algorithm that, given the input, would produce the same or similar output.

The process which led to the construction of the model of expertise presented in this article roughly consisted of the following steps:

INITIAL OBSERVATIONS

Firstly, the protocol was used to make some initial observations about the nature of the task, e.g.:

- What kind of task is it: analytic, synthetic?
- Are there clearly identifiable sub-tasks?
- What can be said about the information and the knowledge that the expert uses?
- Does the task resemble some known (generic) task? If so, what are the similarities and differences?
- Does it seem feasible to automate (part of) the task?

TENTATIVE DOMAIN SCHEMA

Subsequently, a first sketch was made of the types of domain knowledge that play a role in solving this task. This characterization of domain knowledge is done *before* any detailed model construction for a dual purpose:

1. To guide and verify the process of model selection and/or decomposition: is the knowledge available for achieving this task?
2. To prevent as much as possible domain knowledge just being specified because it is required by the particular problem-solving method that was chosen to achieve the task.

   The chosen problem-solving method will of course influence the required representation of domain knowledge. Our goal is, however, to specify such representations as much as possible as a *viewpoint* on the available domain knowledge. For example, in the office-assignment domain relations exist between particular employees and their roles in the department (*employee X has the role of head-of-group*). The fact that this relation can be used as *classification* knowledge is a method-(or use)-specific viewpoint.

MODEL SELECTION AND TOP-DOWN MODEL CONSTRUCTION

The next step was to specify the top-level task (in this case office assignment) in terms of sub-tasks and primitive inferences required for solving the problem. This model construction process consists of one or both of the following activities:

1. Selection of a predefined generic decomposition in sub-tasks and inferences: an interpretation model. The selection of such a model is guided by characteristics

of the task such as the nature of the input and output of the top-level task (e.g. an enumerable set of solutions suggests an interpretation model for an analytic task) and of the required types of domain knowledge (e.g. a model of the normal behaviour of a device). These selection criteria are represented in the form of a decision tree (Wielinga *et al.*, 1992: p. 32).

2. A (repeated) process of model decomposition. In the worst case, no (partial) interpretation model is available for the task at hand. The knowledge engineer then has to decompose the top-level task into sub-tasks and inferences (primitive leaf tasks) on the basis of the elicited data (in particular protocols).

   There are, however, also a number of other situations in which decomposition plays a role:

   - *The top-level task is not a generic task for which an interpretation model can be selected, but is a compound, "real-life" (Breuker et al., 1987), task.* In that case, the knowledge engineer will first have to decompose the top-level task to the level of generic tasks.
   - *The decomposition given by the selected interpretation model is too coarse-grained.* The "inferences" in such a model are in fact complex sub-tasks that need further specification and decomposition to arrive at inferences that can be linked to fragments of domain knowledge. For example, many models for synthetic tasks in the KADS library of interpretation models provide only a first level of decomposition.† Also, even if a detailed interpretation model such as systematic diagnosis is selected, it is possible that this model needs further detailing for the task-domain at hand.

Often, there is an interplay between the selection of generic components and model decomposition. In the office assignment case the emphasis was on decomposition, as there was no detailed interpretation model available.

REFINEMENT

When a first (partial) model of expertise has been established through selection and/or decomposition, it will need to be refined. This refinement was in this case performed in two ways:

1. By formulating task structures (i.e. control relations between sub-tasks) and checking whether these task structures could serve as plausible explanations of the behaviour of the expert.
2. By trying to identify the types of domain knowledge that would be needed to carry out the various inferences, and checking whether this knowledge could be derived from the domain schema. If it is not derivable, the question arises whether it can be formulated as an extension of this theory and whether expertise data are available for formulating this knowledge. Often, this involves additional knowledge elicitation (KE goal).

The refinement process acts in a sense as a verification of the chosen decomposition.

† One could view problem-solving methods such as "propose & revise" (Marcus & McDermott, 1989), "cover & differentiate" (Eshelman, 1988) and "skeletal planning" (Musen, 1989) also as partial interpretation models that can be used as a starting point for a model of expertise.

In the next section, the initial observations about the office assignment problem are discussed.

3. Initial observations

Initially, the protocol is our focus of attention. While reading the protocol, we noted the following features of the problem-solving process of the expert:

- A first thing to note is that the office assignment problem is of a synthetic nature: the solution is not chosen from a given set of predefined solutions, but is constructed using knowledge about employees, rooms and allocation constraints.
- The expert appears to solve the problem in two steps: (i) *selecting* a particular (group of) employee(s), and (ii) *assigning* this (group of) employee(s) to a room.
- It seems that the selection process is based on a global plan of the expert, namely assigning employees in a particular order. This plan is, however, not explicitly mentioned by the expert. This assumption would need to be verified in a future session with the expert (KE goal).
- The ordering in the allocation plan is not an ordering of specific employees, but of *types* of employees, e.g. head of group, manager, etc. The underlying knowledge on which this ordering is based seems to be quite subtle. For example, it is not just based on a simple hierarchy of employee types, as one could be inclined to deduce from the fact that the head of group is assigned first: this would not explain why the secretaries are assigned before the manager and the heads of projects.
- The elements of the allocation plan are not just single employees. These elements can also be sets of employees that are assigned in a random order (heads of projects) or groups of employees that are assigned in blocks to a room (secretaries, researchers).

  If one requires of the final model that it indeed models the behaviour of the expert as closely as possible, then this would exclude every model or method in which employees are assigned one at a time.
- The expert does not backtrack in the protocol. There is no evidence of a verification and/or a revision process. Most existing models and systems for synthetic problems (e.g. Marcus & McDermott, 1989; Chandrasekaran, 1990) contain such a verify/revise step. The absence of this step could very well be an artifact of the sample problem solved in the protocol. This should be a major topic for future sessions with the expert (KE goal). We will come back to this issue in the discussion section.

This is by no means meant to be a complete or even correct list.† Such initial observations, however, focus the modelling process (see the next section).

In the next three sections, the results of the process of modelling expertise are

† It is in fact the list that the author presented at the EKAW'91 Sisyphus workshop in Crieff, Scotland, after a first reading of the sample problem.

described, notably:

- Description of the domain schema (Section 4).
- Classification of the office-assignment task and model selection (Section 5).
- Model decomposition and resulting inferences and tasks (Section 6).

4. Domain schema

In the description of the domain knowledge we are mainly interested in a structural description: what types of knowledge are available in the problem description? For this schematic description we use the constructs of the KADS data-description language (DDL) (Schreiber *et al.*, 1993: chapter 4): concepts, sets, properties, and relations between concepts, instances and/or expressions. Figure 1 gives a graphical overview of the structure of the domain knowledge described below.

Employees and rooms stand out as central concepts in this domain. Employees have properties (such as whether they smoke or like to hack) and relations with projects they work on or are the head of. Also, a number of relations between two employee instances seem to be important: a smoker and a non-smoker, employees working on different projects, etc. Rooms have a number of properties (size, number, type, etc.) and relations with other rooms (distance, next to).

A DDL description of the concept *employee* and of one relation between employee

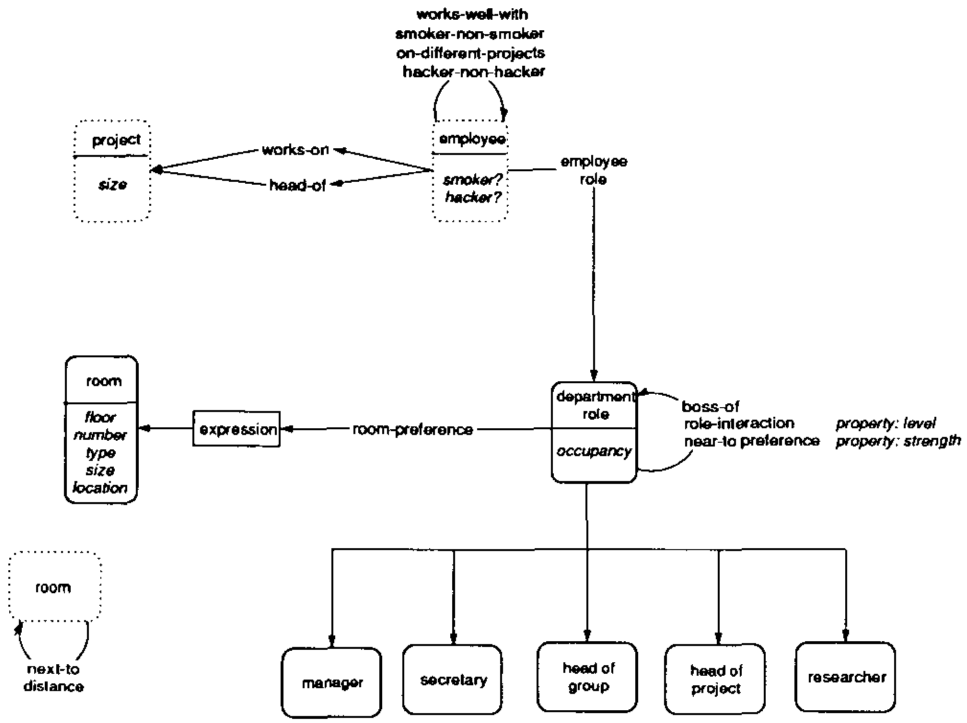

FIGURE 1. Schema of the domain knowledge in the office assignment domain. See Schreiber *et al.* (1993: chapter 4) for an explanation of the graphical notation used.

instances is given below. A full DDL description of the domain schema is given in Schreiber (1992: appendix A).

concept *employee;*
  properties:
```
  smoker: [true, false];
  hacker: [true, false];
```

relation *on-different-projects;*
  argument-1: `instance(employee);`
  argument-2: `instance(employee);`
  semantics: `associative;`
  axioms:
```
    ∀E1, E2:employee, P1, P2:project
     on-different-projects(E1, E2) ↔
     works-on(E1, P1) ∧ works-on(E1, P1) ∧ P1 ≠ P2;
```

Another central concept in this domain is the notion of a *department role*: head of group, secretary, etc. As observed in the previous section, the expert seems to base most of his allocation decisions on properties of employee *types* and not on individual employees. The employee types are represented as sub-concepts of department role (see Figure 1).

Several types of relations concerning department roles seem to be important in the domain:

- A hierarchy of roles (e.g. the head of the group is the boss of the manager);
- The amount of daily interaction (e.g. a high level of interaction between the head of the group and secretary);
- Positional preferences (e.g. the head of the group should be near to a secretary);
- Relations between department roles and expressions about rooms, denoting room preferences (e.g. the head of the group should have a large, central room).

This room-preference relation is represented in the DDL as follows:

relation *room-preference;*
  argument-1: `department-role;`
  argument-2: `expression(room);`
  tuples:
```
    ⟨department-role, type(room) = office⟩
    ⟨head-of-group, location(room) = central⟩
    ⟨head-of-group, size(room) = large⟩
    ⟨head-of-project, size(room) = small⟩
    ⟨researcher, size(room) = large⟩
    ⟨manager, size(room) = small⟩
    ⟨secretary, size(room) = large⟩;
```

These relation tuples should be interpreted as universally quantified statements about the employees that fulfil a particular role, e.g. the statement about "head of project" should be interpreted as "all heads of projects need to get some small, single room".

5. Task classification and model selection

The office-assignment task takes as input a set of employee instances and a set of room instances and produces as output a set of allocations of rooms to employees. The office-assignment task can be classified as a design task: although the solutions are in principle enumerable for a given input problem, in practice the solution is not selected, but constructed.

In Chandrasekaran (1988) three classes of design tasks are described: creative design tasks, routine design tasks, and a mix of routine and creative design. The prime property of routine design is that the elements from which the solution is constructed are known in advance. Office-assignment can thus be classified as a routine design task.

Puppe (1990) distinguishes three sub-classes of routine design tasks: planning, configuration and allocation (in German: "zuordnung"). According to Puppe, the main features that distinguish allocation from planning and configuration are:

- It operates on at least two disjunct sets of objects.
- The solution consists of allocation relations between objects of different sets that satisfy particular requirements.

Office-assignment is thus clearly an allocation task. The two disjunct sets of objects are in this case the employees and the rooms.

Unfortunately, the KADS interpretation model library in Breuker *et al.* (1987) does not contain a model for allocation. In such a case, it can be useful to look at a more general model for design tasks and use this as a starting point. Such a model for a more general task provides, however, only a first level of decomposition.

Chandrasekaran (1990) describes methods for routine design tasks. The general structure of the design task is presented as consisting of three major sub-tasks: propose, critique and modify. For each sub-task a number of methods are described (informally) that can be used for realizing the task. For example, the propose task can be realized with decomposition methods, with constraint satisfaction, etc.

The SALT system (Marcus & McDermott, 1989) implements a similar model for routine design called "propose & revise". The propose step proposes a value for a design parameter. Design parameters are linked to design constraints. When a constraint violation is detected, the revise task is activated to suggest changes ("fixes") to the design. This process is iterated until all design parameters have a value and no constraints are violated.

In the mixer-configuration system (Wielemaker & Billault, 1988), design starts with building an ordered list of "duties" (i.e. design requirements). The first duty of the list (the "top duty": the requirement which is considered to be the most critical one) is used to generate an initial configuration, which is subsequently tested and refined on the basis of the other requirements. If a conflict arises, for example because some requirement cannot be satisfied, this duty becomes the top-duty and the design is modified.

In each of these models, the general structure of routine design appears to have an iterative structure: first, a (partial) solution is proposed, which is subsequently verified and if necessary adapted and/or refined. This leads to a new proposal and thus starts a new cycle of verification and adaptation/refinement.

As noted in Section 3, the expert in the sample protocol seems to carry out only the *propose* task. We limit the modelling enterprise in this article to a study of the nature of this propose task. However, this apparent absence of verification and revision should be a major focus for further knowledge engineering.

6. Task and inference knowledge

Initially, we observed (Section 3) that this propose task seems to consist of two steps: selecting an employee and assigning him/her to a room. Also, the point was made that this selection step seemed to be based on a global allocation plan. In other models for design tasks the notion of a plan also appears. For example, in the mixer-configuration system (Wielemaker & Billault, 1988), the notion of a plan plays a role in terms of an ordering of requirements. "Tackle the most difficult requirement first" appears to be a quite general strategy in design tasks. We will assume here that the expert indeed has some allocation plan. The precise nature of this plan is discussed below. As already pointed out, this assumption would need to be verified during further knowledge engineering (KE goal).

This gives us a first decomposition of the propose task (see Figure 2). This figure should be interpreted as a *provisional* inference structure. It fulfils 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 and knowledge differentiation (see also Schreiber *et al.*, 1993: chapter 5).

As task and inference knowledge are described in a domain-independent way, we coin the general role names *component* and *resource* to talk at the task and inference

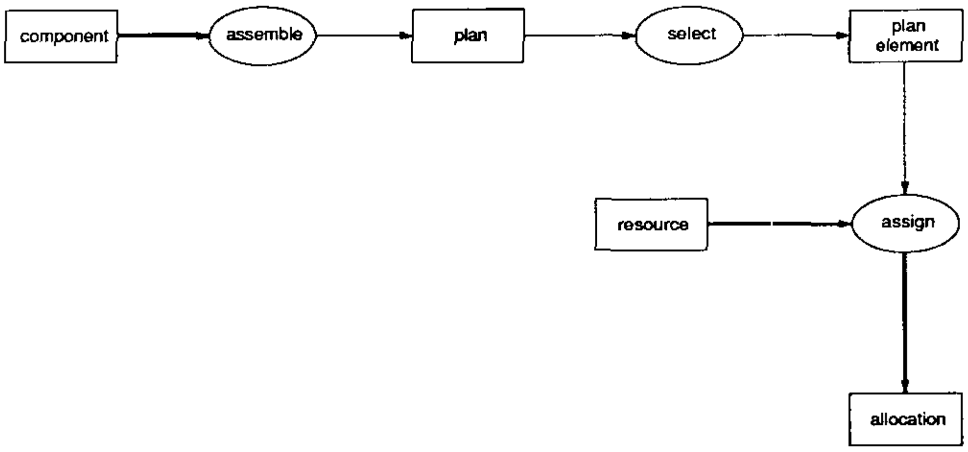

FIGURE 2. First provisional structure of the propose task. See Schreiber *et al.* (1993: chapter 5) for a description of the graphical notation used.

level about employees and rooms. This is one way of enabling a potential reusage of (part of) the resulting model for another resource allocation domain.

The task structure of the propose task is specified below in a structured-English format. The top-level task *propose-allocations* consists of two major steps:

- *Assemble plan:* which generates a plan in which the allocation order of components is specified;
- *Assign resources:* which produces parts of the solution. This last step is carried out for each element in the plan.

task *propose-allocations*

input:
```
    components: set of components to be allocated
    resources: set of available resources
```
output:
```
    allocations: set of tuples ⟨resource, set of
      components⟩
```
control-terms:
```
    plan: list of (sets of) components representing an
      allocation ordering
    plan-element: (set of) components representing an element
      of the plan
```
task-structure:
```
    propose-allocations(components + resources →
      allocations) =
      assemble(components → plan)
      FOREACH plan-element ∈ plan DO
        assign-resources(plan-element + resources →
          allocations)
```

We use the format proposed in Wielinga *et al.* (1992). The slots *input, output* and *control-terms* describe the data manipulated by the task, such as single objects, tuples, sets and lists. The *task structure* specifies the sub-tasks and their control dependencies in the form of a piece of pseudo-code. The arrows in the task structure describe the relation between input and output of the task or sub-task.

6.1. PLAN ASSEMBLY

The question now arises of whether it is possible to identify one inference that can generate the plan, or whether plan assembly should be considered a non-primitive task that requires further decomposition. To resolve this question we turn back to the protocol.

We noted the following characteristics of the way in which the expert orders the assignment of components:

1. The ordering is not based on individual components, but on *component types*: the expert does not talk about specific employees, but about the head of group, the secretaries, etc. This means that during plan assembly it is necessary to classify components (the input of the assembly task) in terms of component *types*.

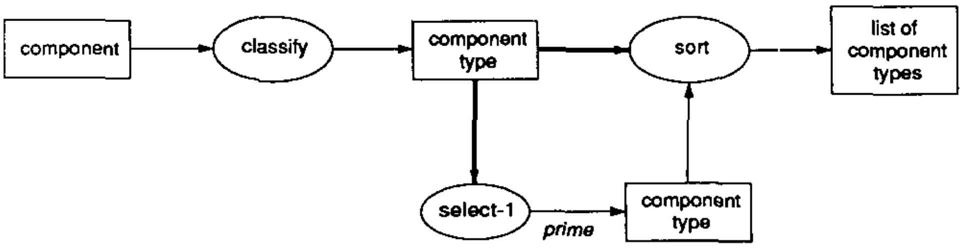

FIGURE 3. Inferences for plan assembly.

2. The head of the group is assigned first, because this assignment "restricts the possibilities of subsequent assignments" (Comment 1b of the protocol). There is a similarity here with other models of design tasks, such as the model of the mixer configuration system (Wielemaker & Billault, 1988): the component which is expected to impose the heaviest constraints on the final solution is tackled first. The allocation plan represents an implicit ordering of requirements: it is not the requirements themselves that are ordered, but the component types to which they are related.
3. The other component types are ordered on the basis of the level of required access to and interaction with the head of group (Comments 2–4).

These observations led us to the formulation of three inferences that are needed to carry out the plan assembly task:

- *Classify* components as component types.
- *Select* the component type with the highest associated constraints.
- *Sort* the other component types relative to the one that imposes the highest constraints.

These three inferences are described in more detail below. The inference structure in Figure 3 shows the dependencies between the inferences for plan assembly. An important point of the specification of inferences is to indicate for each inference how its functional terms (meta-classes, domain view) relate to available domain knowledge. This will often reveal that some type of domain knowledge is lacking and can thus lead to new KE goals.†

Classify
The *classify* knowledge source uses the domain relation *employee-role* (see Figure 1) for classifying a component (an employee instance) as a component type (i.e. a department role).

knowledge-source *classify*
  input-meta-class
    `component` $\rightarrow$ `employee`
  output-meta-class
    `component-type` $\rightarrow$ `department-role`

† Here we will only describe inferences that use knowledge described in Figure 1, but it is fair to say that this is an artifact of a *post hoc* description.

domain-view
```
    type associations from component to component-type→
      employee role(employee, department-role)
```
description
```
    knowledge-base look-up
```

The arrows in the description above show how names at the inference level map onto domain terms. The meta-classes can be seen as the data elements that are being manipulated by the knowledge source. The domain view describes the static knowledge that is used in this inference. The "description" slot gives an indication of how the output could be generated from the input and the domain view. This allows the knowledge engineer to make some remarks about possible computational methods. The actual selection of a computational method (which could turn out to be a different one) is part of the design process (see Section 7).

Select prime
The *select-1* inference is used in the plan-assembly task to find the component type with the highest requirements. This knowledge source uses a domain relation *boss-of* to find the highest node in the component-type hierarchy.† This component (for which we will use the term "prime") is assumed to be the most critical one to assign (Comment 1 of the protocol).

knowledge-source *select-1* (select prime)
input-meta-class
```
    component-types→set of department-role
```
output-meta-class
```
    prime→department role
```
domain-view
```
    hierarchy of component-types→
      boss-of(department-role, department-role).
```
description
```
    find the top node in the hierarchy of component types
```

Sort
As remarked in Section 3, the other components are sorted on the basis of the amount of interaction that is required between certain types of components (see Comments 2–4 of the protocol).

knowledge-source *sort*
input-meta-class
```
    prime→department-role
    components-types→set of department-role
```
output-meta-class
```
    component-types→list of department-role
```

† In retrospect, this is probably a sub-optimal specification, because it makes unnecessary strong assumptions about the nature of the domain knowledge. It is conceivable that in other tasks other types of domain knowledge than hierarchies could be used to select the component with the highest associated constraints.

```
domain-view
    sort predicate→
      value of the attribute ''level'' of the relation
      role-interaction(department-role,
        department-role)
description
    a component type is placed before another
    component type if the level of required interaction
    with the prime is higher
```

Plan assembly tasks

In the task-knowledge specification for the plan assembly task we have to indicate how the three inferences can be sequenced to achieve the goal of the task: the construction of a plan. The simplest solution would be to specify one task structure for plan assembly. However, the select and sort inferences are so tightly connected that we decided to view this part as a separate sub-task *order*. A reason for this more detailed task decomposition is that one can envisage that in other domains this task could be realized with one inference.†

We thus end up with three tasks that specify the sequencing of inferencing in plan assembly: plan assembly and two sub-tasks: (i) a classification task, and (ii) an *ordering task.*

The plan-assembly task is specified as follows:

```
task assemble-plan
  input: components
  output: plan
  control-terms:
      component-types: set of components classes
  task-structure
      assemble-plan(components→plan) =
        classify(components→component-types)
        order(component-types→plan)
```

The *classify* task requires a repeated invocation of the *classify* knowledge source plus a data operation (set unification).

```
task classify
  input: components
  output: component-types
  task-structure:
      classify(components→component-types) =
        FOREACH component ∈ components DO
          classify(component→component-type)
          component-types := component-type ∪ component-
            types
```

† Although this may sound a bit altruistic, the whole idea of "model construction for reusability" is so central to the KADS approach that it tends to become second nature for people involved in it.

For readability purposes, the names of knowledge sources are italicized in the task structure.

The *order* task specifies a sequence of the select and sort inferences and appends the output of both inferences to the resulting allocation plan.

```
task order
  input: component-types
  output: plan
  control-terms:
      prime: the component-type with the highest
        constraints
      other-components: the components minus the prime
        component
      ordered: the other components sorted with respect to
        constraints in relation to the prime
  task-structure
      order(component-types→plan) =
        select-1(component-types→prime)
        other-components := component-types/prime
        sort(other-components + prime→ordered)
        plan := prime, ordered
```

The "/" symbol represents a subtraction operator on a set; the "," symbol is used here to specify the order in a list. The resulting plan consists of an ordered list of component types.

6.2. ASSIGN RESOURCES

In the *assign-resources* task, components of one particular type are allocated to a resource. Again, we turn to the protocol to study the inferences involved in assigning resources.

- As was noted in Section 3, if it concerns a multiple assignment (more than one component to one resource) the expert first groups these components into units of the right size using a special type of requirement concerning component interaction (avoiding conflicts and enhancing synergy, see protocol Comments 7–10). The type of assignment (single or shared) is fully determined by the component type (e.g. a head of a project should have a single room).
- The input for the actual *assign* task with respect to the components to be allocated can be of two types (see the remarks in Section 3):

  1. One single component (head of group, manager) or component group (secretaries).
  2. A *set* of components (heads of projects) or component groups (researchers).

  If the input is a set, the assignment order of its elements should be random, as the expert indicates in the protocol explicitly that there is no particular reason for his sequencing of, for example, assignments of heads of projects and pairs of researchers (Comments 4–6 and 8).

These observations lead us to a first refinement of *assign resources* by introducing an additional *group* step. This refined structure of the *assign* step in Figure 2 is shown in Figure 4, which again should be interpreted as a *provisional* inference structure.

```
  task assign-resources
    input:
        component-type: type of component allocated in this
          plan step
        resources: available resources
    output:
        allocations
    control-terms:
        unit: a component or set of components
        grouping: set of units
        suitable-groupings: groupings satisfying particular
          constraints
task-structure
        assign-resources(component-type + resources→
          allocations) =
          group(component-type→suitable-groupings)
          select-random(suitable-groupings→grouping)
          REPEAT
            select-random(grouping→unit)
            assign(component-type + unit + resources→
              allocations)
            UNTIL grouping = ∅
```

The *group* step is only interesting for components that share resources. For other component types we assume it is an empty operation. The random selection of units in the REPEAT loop ensures that, for example, heads of projects are really assigned in a random order. This also implies that the specification differs here slightly from the assignment order in the protocol. There, a unit of two researchers is assigned directly after grouping. As the expert indicates that there is no special reason for this (except maybe mental hygiene) we have separated in our model the grouping of units from the actual assignment of units.

It might be useful to note that the introduction of a separate *group* step implies a differentiation of allocation requirements into two major types: (i) requirements concerning interaction of components with respect to one resource (conflicts, etc.), and (ii) resource-specific requirements (room preferences, etc.). This is in fact a role differentiation at the level of the model of expertise that can make the resulting system more efficient (Schreiber, Akkermans & Wielinga, 1991). For example, a computational technique implementing one of these sub-tasks would need to handle less requirements and operate on a smaller set of components (because some of them are already grouped into units).

In the following sections we study the *group* and the *assign* step in more detail.

6.3. GROUP

When components are grouped together for joint assignments to one resource, a different kind of domain knowledge comes into play, namely knowledge about possible effects of the joint usage of the resource. The expert tries to minimize negative effects and support positive ones as much as possible. This grouping of components into appropriate units (Comments 7–10 in the protocol) appears to be the only part of the resource allocation process where the expert uses requirements based on properties of *individual* employees, e.g. whether they smoke or on which project they work, etc.

Generating suitable groupings is typically a task where one would specify a different problem-solving method for a machine than the one the expert employs. The expert generates in the protocol partial groupings based on the requirements. This partial grouping is in fact one out of a set of possible partial groupings. Given the limited size of human short-term memory it is usually impossible to consider all possible solutions. For a machine however, this storage problem does not exist. On the other hand, the somewhat *ad hoc,* intuitive way in which the expert generates a grouping would be quite difficult to model for machine execution.

Thus, we decided to drop for this subtask the general guideline of modelling the expert as closely as possible and model the grouping task as consisting of two types of inferences:

- A *transform* inference, which generates all possible groupings.
- A *select* inference, which selects a subset of groupings that satisfies particular requirements.

The transform inference is described below:

knowledge-source *transform*
input-meta-class:
```
    component-type→department-role
```
output-meta-class:
```
    possible groupings→set of employee structures
```
domain-view:
```
    –
```
description:
```
    generate all possible groupings of components of this
    type
```

For the select inference ("select-2", to distinguish it from the previous select inference), one has to decide what the requirements should be for suitable groupings. The following types of requirements are mentioned by the expert:

- *Conflicts:* The expert tries to minimize conflicts. Putting a smoker and a non-smoker together is considered a major conflict (Comment 7) and should have a high impact. Putting a hacker and a non-hacker together is only a minor conflict that could be allowed if more important reasons exist for preferring such a grouping.
- *Synergy:* The expert also tries to maximize synergy. Putting employees together who work on different projects is considered by the expert to be an important

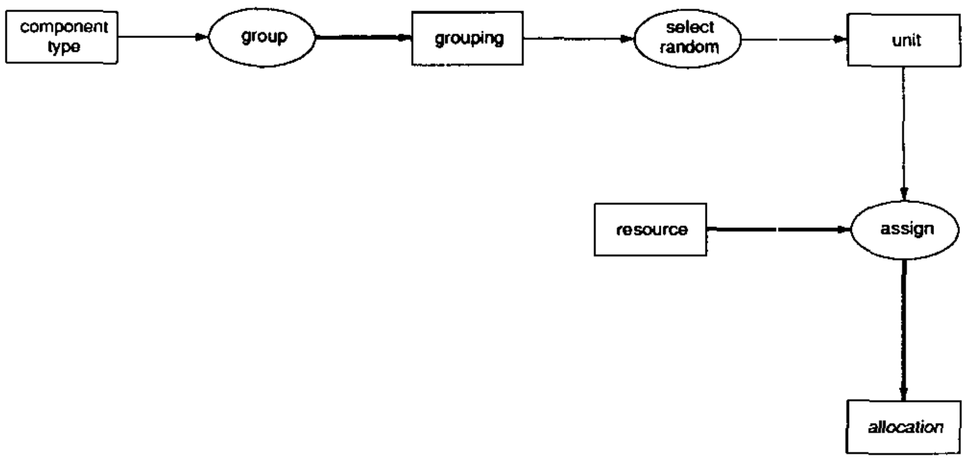

FIGURE 4. First refinement of the *assign* step of Figure 2 by introducing a group step which generates possible groupings and a random selection of a unit (a component or set of components to which one resource will be assigned).

type of synergy (Comment 8). Also, grouping researchers working on similar subjects is considered synergetic, although to a lesser degree (Comment 10).

The select inference specifies the selection of a subset of groupings given one particular criterion (some conflict or synergy). Based on the observations above, we distinguish four types of criteria: minimize major/minor conflicts and maximize major/minor synergy. This choice would have to be verified in future sessions with the expert (KE goal).

The dependencies between these two inferences, which constitute a refinement of the *group* step in Figure 4, are shown in Figure 5.

knowledge-source *select-2 (select suitable groupings)*
input-meta-class:
```
    groupings→set of employee structures
    selection-criterion→a conflict- or synergy-type
```
output-meta-class:
```
    suitable-groupings→set of employee structures
```

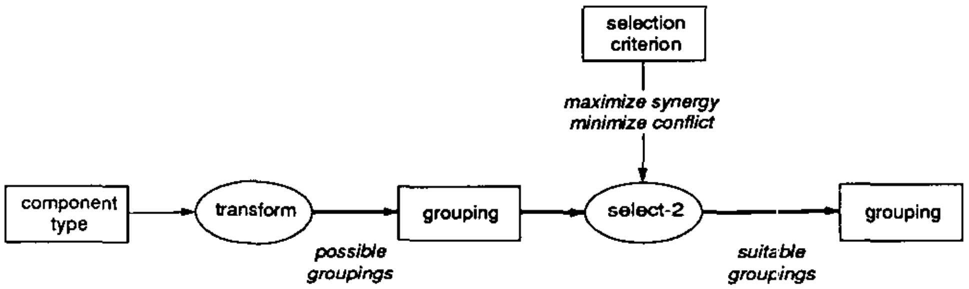

FIGURE 5. Inferences for generating suitable groupings of components.

domain-view:
```
    major-conflict→smoker-and-non-smoker relation
    minor-conflict→hacker-and-non-hacker relation
    major synergy→on-different-project relation
    minor-synergy→works-with relation
```
description:
```
    generate the subset of all possible groupings
    that minimizes some conflict or maximizes some type of
    synergy.
```

This distinction between four different types of criteria can be considered as an example of *inference differentiation* (cf. Schreiber *et al.*, 1993: chapter 5): the *select-2* inference can be differentiated into four sub-types, each using a different criterion.

In the task-knowledge specification of *group* we have to decide in which order these four possible instantiations of the *select-2* inferences should be executed. Looking at the protocol, we decided that the order "avoid major conflict, increase major synergy, increase minor synergy, avoid minor conflict" conforms most to the way in which the expert solves the grouping problem. Again, this hypothesis would need to be verified (KE goal).

task *group*
input:
```
    component-type: the type of components being grouped
```
output:
```
    preferred-groupings: the optimal sub-set of groupings
      given the selection criteria
```
control terms:
```
    possible-groupings: all possible groupings of the
      components of this type
```
task-structure:
```
group(component-type→suitable-groupings) =
  transform(component-type→possible groupings)
  select-2(possible-groupings+minimize(major-
    conflict)→preferred-groupings)
  select-2(preferred-groupings+maximize(major-
    synergy)→preferred-groupings)
  select-2(preferred-groupings+maximize(minor-
    synergy)→preferred-groupings)
  select-2(preferred-groupings+minimize(minor-
    conflict)→preferred-groupings)
```

See Section 7 for a sample system trace of the execution of this task.

6.4. ASSIGN

In the assign task resources are allocated to components or groups of components on the basis of various requirements. We distinguished two types of such

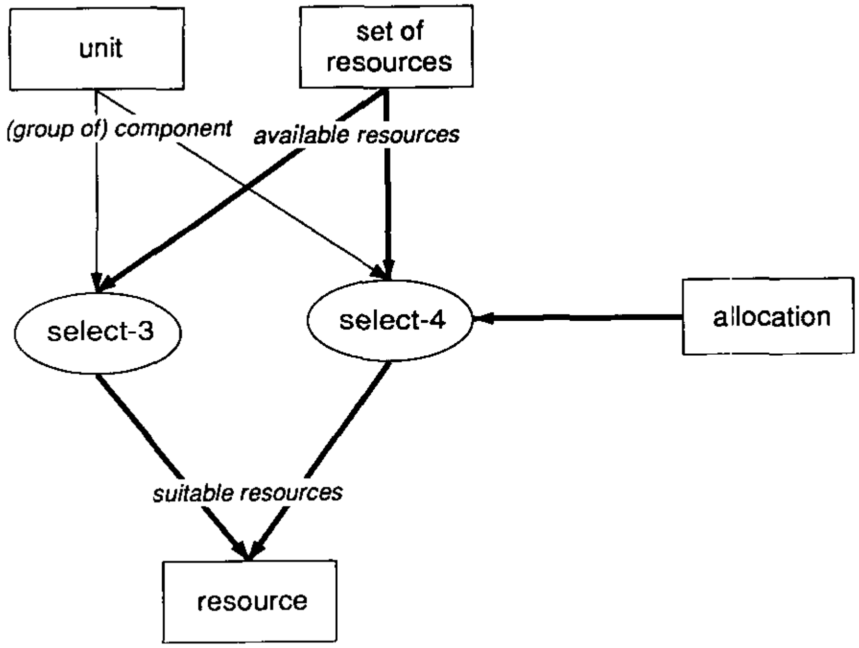

FIGURE 6. Inferences for resource selection.

requirements:

1. *Resource specific requirements*: Requirements about a resource independent of other allocations: required size, required location, etc.
2. *Positional requirements*: Requirements about a resource that are *dependent on other allocations*, e.g. a room is required as close as possible to the head of group.

These requirements are the same for every component of a particular type.

Thus, we defined two select inferences, *select-3* and *select-4*, each selecting a subset of resources that respectively satisfy resource-specific and positional requirements. The *select-4* inference has as an additional input the current set of allocations.

Figure 6 shows the dependencies between the two *select* inferences. This figure represents a further detailing of the *assign* step in Figure 4.

knowledge-source *select-3 (select on resource requirements)*
input-meta-class:
```
component-type→department-role
resources→set of rooms
```
output-meta-class:
```
suitable-resources→set of rooms
```
domain-view:
```
resource-requirement→room-preference relation
```

description:
```
    select the subset of resources that satisfies
    resource-specific requirements
```

knowledge-source *select-4 (select on positional requirements)*
input-meta-class:
```
    component-type → department-role
    resources → set of rooms
```
output-meta-class:
```
    suitable-resources → set of rooms
```
domain-view:
```
    resource-requirement → near-to-preference relation
```
description:
```
    select the subset of resources that satisfies
    positional requirements
```

Note that the unit to which a resource will be assigned is input to neither of the two inferences. This is consistent with the fact that resources are only selected based on requirements connected to a component *type*. The main decision that has to be taken when defining control over these inferences is which one should be executed before the other (or maybe in parallel)? In the current task structure *select-3* is executed before the *select-4* inference. This implies that we give a higher priority to resource-specific requirements. If, after execution of both inferences, more than one resource is considered suitable, one is selected at random.

task *assign*
input:
```
    component-type:
    unit: the component or group of components that to
      which a resource is assigned
    resources: available resources
    allocations: current allocations
```
output:
```
    resources: available resources
    allocations: current allocations
```
control-terms:—
task-structure:
```
    assign(⟨component-type + unit + allocations +
      resources → allocations + resources) =
      select-3(component-type + resources → suitable-
        resources)
      select-4(component-type + suitable-resources +
        allocations → suitable-resources)
      select-random(suitable-resources → resource)
        allocations := ⟨unit, resource⟩ ∪ allocations
      resources := resources / resource
```

A sample system trace of the execution of this task can be found in Section 7.

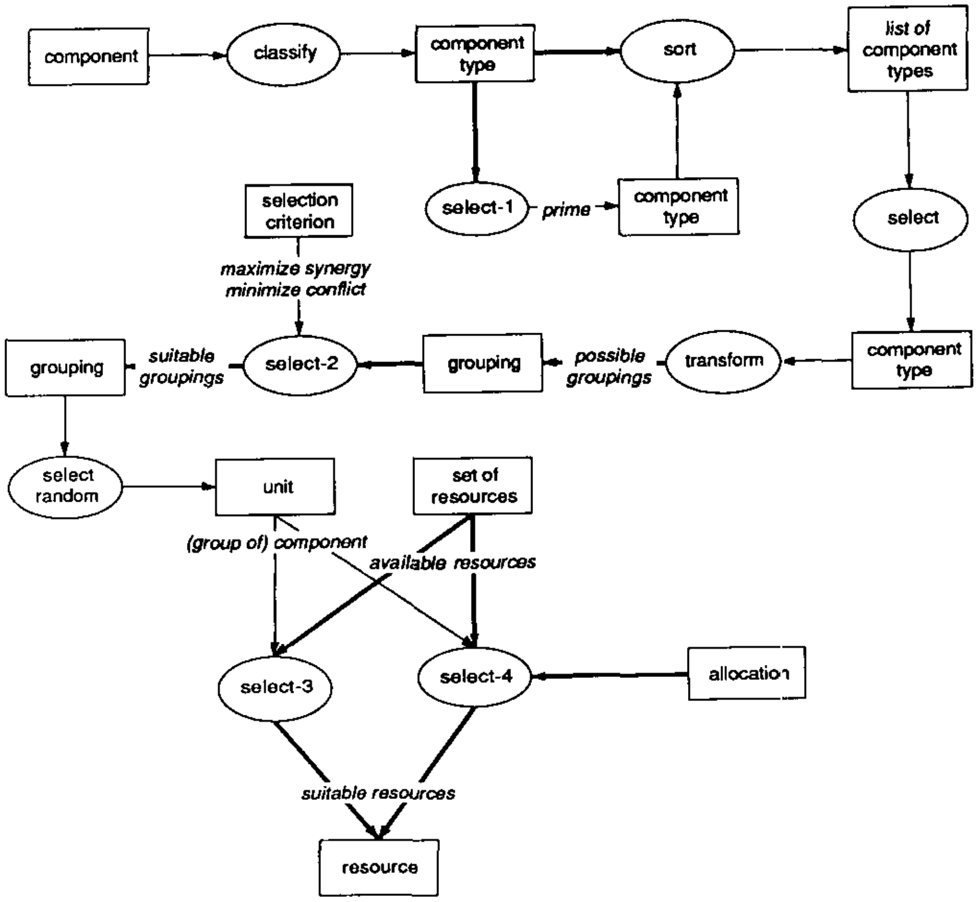

FIGURE 7. Inference structure for resource allocation in the office-assignment domain. The figure summarizes the results of the various decompositions and refinements of the first model in Figure 2.

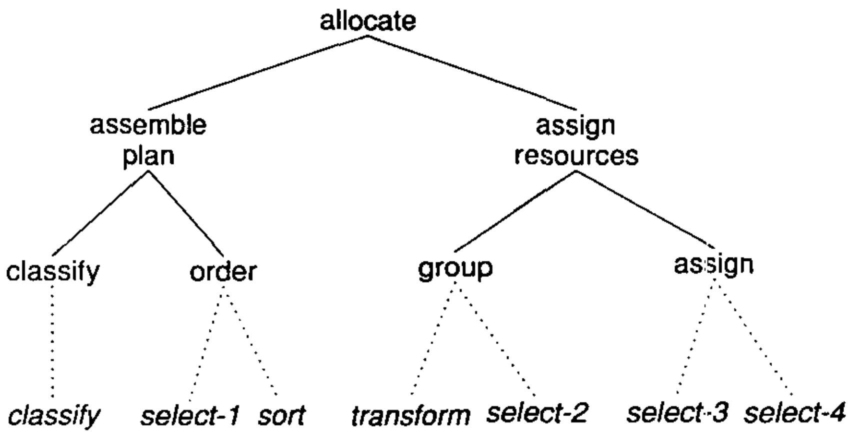

FIGURE 8. Task decomposition of the office assignment problem. Italic names denote knowledge sources. Two trivial *select* inferences (*select-next* and *select-random*) have been left out.

The full inference structure that resulted from the model construction process for this model of resource allocation is shown in Figure 7. Figure 8 shows the resulting task decomposition.

7. Resulting system

In this section we describe some aspects of the design and implementation of a system that implements the behaviour specified in the model of expertise. In the design of the system we follow the structure-preserving principle (Schreiber *et al.*, 1993: chapter 6): all relevant elements of the conceptual model should map onto clearly identifiable constructs in the system. The advantages of such a design approach are:

- It simplifies the implementation of an explanation facility that enables the user and/or the expert to trace the system's execution in the vocabulary of the model of expertise. Although we have not built a graphical interface for this particular case, we have tried to ensure that all the necessary anchor points for such an extension are present.
- It provides clear routes for refining and/or extending the system, such as:
  - Adding/modifying domain knowledge, e.g. other conflicts or room requirements.
  - Changing the control of task execution.
  - Replacing computational techniques.
  - Introducing additional tasks and inferences such as for verification and revision.

No special-purpose tools were used in the development of this system. Also, the fact that no run-time interaction with external agents such as a user is required simplified the system development. The chosen environment was the SWI-Prolog system (Wielemaker, 1991), mainly for pragmatic reasons. The system architecture is an instantiation of the skeletal architecture advocated by KADS (Schreiber *et al.*, 1993: chapter 6). Modules were used to support the separation of various elements of this architecture. The system allows the user to trace system execution optionally at the *task level,* displaying activation and termination of tasks, together with the corresponding input and output, and/or at the *inference level,* displaying the results of the execution of inferences. All system output is plain text. No fancy user interface was developed.†

In the rest of this section, some partial traces of the system are shown to illustrate how the system solves the problem with the given data sets. The source code of the application plus a full execution trace for the first Sisyphus problem can be found in Schreiber (1992: appendix B).

7.1. SISYPHUS-91 PROBLEM

Figure 9 shows a task-level trace of the initial activation of the *propose task* and the subsequent generation of the allocation plan. Input to the *propose* task is the set of components and resources. In the *plan assembly* task (see Section 6.1) the

† The total development time of the system was one week.

```
Activating task "propose allocations"
  input : components = [Werner L.,Marc M.,Angi W.,Juergen L.,Andy
                        L.,Michael T.,Harry C.,Uwe T.,Thomas D.,Monika X.
                        ,Ulrike U. ,Hans W. ,Eva I.,Joachim I.,Katharina M.]
  input : resources = [C5-113,C5-114,C5-115,C5-116,C5-117
                       ,C5-119,C5-120,C5-121,C5-122,C5-123]

Activating task "assemble plan"
  input : components = [Werner L.,Marc M.,Angi W.,Juergen L.,Andy
                        L.,Michael T.,Harry C.,Uwe T.,Thomas D.,Monika
                        X. ,Ulrike U. ,Hans W. ,Eva I.,Joachim I.,Katharina M.]

Activating task "classify"
  input : components = [Werner L.,Marc M.,Angi W.,Juergen L.,Andy
                        L.,Michael T.,Harry C.,Uwe T.,Thomas D.,Monika
                        X. ,Ulrike U. ,Hans W. ,Eva I.,Joachim I.,Katharina M.]

Task "classify" terminated
  output: component types =
          [manager,head_of_group,head_of_project,secretary,researcher]

Activating task "order"
  input : component types =
          [manager,head_of_group,head_of_project,secretary,researcher]

Task "order" terminated
  output: allocation plan =
          [head_of_group,secretary,manager,head_of_project,researcher]

Task "assemble plan" terminated
  output: allocation plan =
          [head_of_group,secretary,manager,head_of_project,researcher]
```

FIGURE 9. Task-level trace of the activation of the *propose* task and the results of *plan assembly* tasks.

components are dynamically classified and subsequently ordered to generate an allocation plan.

In Figure 10 it is shown how the researchers are grouped into units of two, based on the criteria set out in the previous section (see Section 6.3). The *transform* inference generates 105 possible groupings. Avoiding major conflicts reduces this set to 15, by grouping the two smokers Andy and Uwe into one unit. Maximizing synergy by putting people on different projects together reduces this set further to 10 possible groupings. Maximizing synergy by grouping people that work on similar subjects reduces the set of 10 to two groupings. The last inference (reducing hacking conflicts) has no effect in this particular case.

The two groupings generated by the program differ slightly from the grouping generated by the expert in the protocol. This is due to the fact that we assumed that the "works-with" relation in the sample data set represented the notion of working on similar subjects that the expert talks about. Probably, this was not a correct assumption and should be noted as a KE goal. However, this type of refinement does

```
Activating task "group"
  input : plan element = researcher

Invoking inference Transform into possible groupings
  input : component_type "researcher" (a [concept(department_role)])
  output: groupings  < 105 POSSIBLE GROUPIIGS, IOT LISTED >

Invoking inference Select suitable groupings
  input : current groupings < SEE OUTPUT PREVIOUS IIFEREICE >
  input : selection criterion "criterion(minimise, major_conflict)"
  output: subset of groupings < 15 GROUPIIGS, IAMELY PRECISELY THOSE
            II WHICH THE TWO SMOKERS (AIDY AID UWE) ARE GROUPED TOGETHER >

Invoking inference Select suitable groupings
  input : current groupings < SEE OUTPUT PREVIOUS IIFEREICE >
  input : selection criterion "criterion(maximise, major_synergy)"
  output: subset of groupings
"[[[Werner L.,Harry C.],[Marc M.,Angi W.],[Juergen L.,Michael T.],[Andy L.,Uwe T.]],
  [[Werner L.,Harry C.],[Marc M.,Juergen L.],[Angi W.,Michael T.],[Andy L.,Uwe T.]],
  [[Werner L.,Harry C.],[Marc M.,Michael T.],[Angi W.,Juergen L.],[Andy L.,Uwe T.]],
  [[Werner L.,Juergen L.],[Marc M.,Harry C.],[Angi W.,Michael T.],[Andy L.,Uwe T.]],
  [[Werner L.,Juergen L.],[Marc M.,Michael T.],[Angi W.,Harry C.],[Andy L.,Uwe T.]],
  [[Werner L.,Marc M.],[Angi W.,Harry C.],[Juergen L.,Michael T.],[Andy L.,Uwe T.]],
  [[Werner L.,Marc M.],[Angi W.,Michael T.],[Juergen L.,Harry C.],[Andy L.,Uwe T.]],
  [[Werner L.,Michael T.],[Marc M.,Angi W.],[Juergen L.,Harry C.],[Andy L.,Uwe T.]],
  [[Werner L.,Michael T.],[Marc M.,Harry C.],[Angi W.,Juergen L.],[Andy L.,Uwe T.]],
  [[Werner L.,Michael T.],[Marc M.,Juergen L.],[Angi W.,Harry C.],[Andy L.,Uwe T.]]]"
  (a set(structure([instance(employee)])))

Invoking inference Select suitable groupings
  input : current groupings < SEE OUTPUT PREVIOUS IIFEREICE >
  input : selection criterion "criterion(maximise, minor_synergy)"
  output: subset of groupings
"[[[Werner L.,Marc M.],[Angi W.,Michael T.],[Juergen L.,Harry C.],[Andy L.,Uwe T.]],
  [[Werner L.,Michael T.],[Marc M.,Angi W.],[Juergen L.,Harry C.],[Andy L.,Uwe T.]]]"

Invoking inference Select suitable groupings
  input : current groupings < SEE OUTPUT PREVIOUS IIFEREICE >
  input : selection criterion "criterion(minimise, minor_conflict)"
  output: subset of groupings
"[[[Werner L.,Marc M.],[Angi W.,Michael T.],[Juergen L.,Harry C.],[Andy L.,Uwe T.]],
  [[Werner L.,Michael T.],[Marc M.,Angi W.],[Juergen L.,Harry C.],[Andy L.,Uwe T.]]]"

Task "group" terminated
  output: groupings =
"[[[Werner L.,Marc M.],[Angi W.,Michael T.],[Juergen L.,Harry C.],[Andy L.,Uwe T.]],
  [[Werner L.,Michael T.],[Marc M.,Angi W.],[Juergen L.,Harry C.],[Andy L.,Uwe T.]]]"
```

FIGURE 10. Trace of inferences involving grouping of researchers. Shortened for readability purposes.

```
Activating task "assign"
  input : plan element = manager
  input : unit = Eva I.
  input : resources = [C5-113,C5-114,C5-115,C5-116,
                       C5-120,C5-121,C5-122,C5-123]
  input : allocations = [[C5-117,Thomas D.],[C5-119,[Monika X.,Ulrike U.]]]

Invoking inference Select on resource requirements
  input :  component_type "manager" (a [concept(department_role)])
  input :  available resources "[C5-113,C5-114,C5-115,C5-116,
                                 C5-120,C5-121,C5-122,C5-123]"
                                (a set([instance(room)]))
  output:  suitable resources "[C5-113,C5-114,C5-115,C5-116]"
                                (a set([instance(room)]))

Invoking inference Select on positional requirements
  input :  component_type "manager" (a [concept(department_role)])
  input :  available resources "[C5-113,C5-114,C5-115,C5-116]"
                                (a set([instance(room)]))
  input :  current positions "[[C5-117,Thomas D.],
                               [C5-119,[Monika X.,Ulrike U.]]]"
                (a structure( [instance (room)], list([instance(employee)])))
  output:  suitable resources " [C5-116]" (a set([instance(room)]))

Working memory operation "select" on "suitable resources"
  with result: "C5-116"
Working memory operation "add [[C5-116,Eva I.]]" on "allocations"
Working memory operation "subtract C5-116" on "resources"

Task "assign" terminated
  output: allocations = [[C5-117,Thomas D.],
                        [C5-119,[Monika X.,Ulrike U.]],
                        [C5-116,Eva I.]]
```

FIGURE 11. Trace of inferences involving assigning a room to the manager.

not affect the overall structure of the model or the system and can be carried out in a later knowledge-refinement phase.

In Figure 11 a sample trace is displayed of the execution of the *assign* task (see Section 6.4) for the manager. In the example, *select-3* generates four suitable rooms, based on the resource requirements (the manager needs a small, single room). *Select-4* then applies the positional requirement of being as close as possible to the head of group and selects from the candidates room C5-116, the one closest to the room allocated to the head of group.

Figure 12 shows the solution generated by the *propose* task in this particular system run. As a number of choices were made random (two possible rooms for the head of group, three possible rooms for the heads of projects and four possible rooms for the researchers; the allocation of the manager and the secretaries is fixed by the allocation of the head of group) the solution space covered by the program includes 288 potential solutions ($2! \times 3! \times 4!$).

```
Task "propose allocations" terminated
  output: allocations =
  [[[C5-117,Thomas D.],[C5-119,[Monika X.,Ulrike U.]],[C5-116,Eva I.],
   [C5-115,Joachim I.],[C5-114,Katharina N.],[C5-113,Hans W.],
   [C5-120,[Andy L.,Uwe T.]],[C5-122,[Marc M.,Angi W.]],
   [C5-121,[Werner L.,Michael T.]],[C5-123,[Juergen L.,Harry C.]]]
```

FIGURE 12. Solution produced by the *propose* task.

7.2. SISYPHUS-92 PROBLEM

The Sisyphus-92 problem introduces a potential constraint violation. A head of project is replaced by a researcher who smokes, leading to an odd number of researchers (nine) and smoking researchers (three). As the problem statement does not contain information about how the expert handles this violation, we had to make some assumptions about how the expert would have solved this new problem. Again, these assumptions would have to be verified (KE goal).

The introduction of a constraint violation raises an interesting point with respect to the model we constructed for the '91 problem. One could argue that the *propose* task should only propose assignments without any constraint violations. This is not true for the model described in this paper: the number of violations is just minimized by the *propose* task. This is in line with the nature of most of the constraints that occur in this domain, e.g. enhancing synergy, reducing minor conflicts, and placing an employee as close as possible to another employee. One can view these constraints as *soft* constraints: the aim is to minimize violations of these constraints, not to reduce such violations to zero. However, there also appear to be *hard* constraints that should in principle not be violated, e.g. a smoking conflict and requirements concerning rooms (size, occupancy).

This hypothesis about a distinction between soft and hard constraints (not mentioned explicitly in the protocol) would need to be verified with the expert and would lead to a refinement of the model of the *propose* task. In particular, the selection criteria used in the grouping task would need to be labelled explicitly as soft or hard constraints. Checking a hard constraint such as smoking would have to produce zero violations (and not just be minimized), otherwise a *revise* task should be invoked. This *revise* task should be able to propose strategies for handling violations, e.g. constraint relaxations. The specification of this *revise* task is outside the scope of this paper, as it would require more detailed information such as a protocol.

Summarizing, the changes needed to handle violations of "hard" constraints require the following changes to the model:

- Introducing an explicit distinction between "soft" and "hard" constraints.
- Invoking the *revise* task when a hard constraint is violated.
- Specification of the *revise* task.

As it happens, the system we developed on the basis of the '91 problem was able to solve the '92 problem after one small modification. Four typical fragments of this

```
Activating task "propose allocations"
  input : components = [Werner L.,Marc M.,Angi W.,Juergen L.,Andy L.,
                        Michael T.,Harry C.,Uwe T.,Thomas D.,Monika X.,
                        Ulrike U.,Hans W.,Eva I.,Joachim I.,Christian I.]
  input : resources =  [C5-113,C5-114,C5-115,C5-116,C5-117,C5-119,
                        C5-120,C5-121,C5-122,C5-123]

.....

Task "group" terminated
  output: groupings = [[[Werner L.,Marc M.],[Angi W.,Michael T.],
                        [Juergen L.,Harry C.],[Uwe T.,Christian I.],[Andy L.]],
                       [[Werner L.,Michael T.],[Marc M.,Angi W.],
                        [Juergen L.,Harry C.],[Uwe T.,Christian I.],[Andy L.]]]

......

Activating task "assign"
  input : plan element = researcher
  input : unit = [Andy L.]
  input : resources = [C5-113]
  input : allocations =
    [[C5-117,Thomas D.],[C5-119,[Monika X.,Ulrike U.]],[C5-116,Eva I.],
    [C5-115,Hans W.],[C5-114,Joachim I.],[C5-120,[Werner L.,Michael T.]],
    [C5-123,[Marc M.,Angi W.]],[C5-121,[Juergen L.,Harry C.]],
    [C5-122,[Uwe T.,Christian I.]]]

......

Task "propose allocations" terminated
  output: allocations =
    [[C5-117,Thomas D.],[C5-119,[Monika X.,Ulrike U.]],[C5-116,Eva I.],
    [C5-115,Hans W.],[C5-114,Joachim I.],[C5-120,[Werner L.,Michael T.]],
    [C5-123,[Marc M.,Angi W.]],[C5-121,[Juergen L.,Harry C.]],
    [C5-122,[Uwe T.,Christian I.]],[C5-113,[Andy L.]]]
```

FIGURE 13. Some sample trace information produced when solving the second (Sisyphus-92) problem. The fragments shown point to the major differences from solving the first problem.

trace are shown in Figure 13. The *group* task generates two suitable groupings (from a set of 945 potential groupings). The groupings are similar to the ones in the '91 problem (see Figure 10), except for the three smokers. Uwe and Christian (the new researcher) are grouped together (because they both like to hack) and Andy is in a group of his own. The modification we made to solve this problem is that incomplete groups (i.e. Andy) are assigned after the complete groups (instead of the random order of unit selection as specified in Section 6.2). Intuitively, this is not an unreasonable strategy. The consequence is that Andy gets assigned to the room that still remains after the other researchers have been allocated, namely the last single room (see the *assign* task activation in Figure 13).

However, it is easy to imagine other constraint violations that would require an explicit revise task, e.g. an odd number of smoking researchers and an even total number of researchers. Thus, it is fair to say that the refinements mentioned earlier would have constituted a more fundamental solution to the '92 problem.

8. Discussion

HOW GENERAL AND/OR REUSABLE IS THE MODEL?

A major assumption in KADS is that the description of task and inference knowledge is sufficiently domain-independent to have the potential of being reused in a similar task domain. With regard to the model that was constructed for the office-assignment domain, the following tentative observations can be made:

- The notion of a plan representing an ordering of requirements seems to be a quite general one: it re-occurs in many constructive task-domains.
- The differentiation into various types of requirements can be useful. In some domains, e.g. allocating air planes to gates, the component-interaction requirements will not be relevant (only one plane per gate), thus leading to a simplified version of this part of the inference structure (the grouping inferences do not have to be included).
- The office-assignment domain contains a number of simplifications that might not be present in other domains and thus may lead to more complex models, e.g.:
  - No time considerations come into play (no existing allocations, no planning of future allocations). This could be very important in a domain such as allocating air planes to gates.
  - Preferences of individual components are not considered in the selection of suitable resources: only preferences of types of components.
- An obvious shortcoming of the model is that it covers only the *propose* task. In most domains, an iterative revision process is required. Some aspects of this revision process were discussed in Section 7. The current model of the *propose* task needs to be refined by making explicit distinctions between soft and hard constraints, and invoking a revision task when hard constraints are being violated. The nature of the revision task would need to be analysed in more detail in order to provide adequate strategies for handling various types of violations.

Concerning the reusability of the domain knowledge, it can be said that the description of employees, rooms, projects, and department roles has a quite general flavour. On the other hand, some relations such as room preferences are rather specific for this task-domain.

COMPARISON WITH OTHER APPROACHES

This exercise has made clear that there is quite some overlap between various approaches to modelling problem-solving. As shown in this article, problem-solving methods described by Chandrasekaran (1988, 1990) and Marcus and McDermott (1989) could be used as input for a KADS modelling enterprise. We see two major differences between the Generic Task approach (as described in Chandrasekaran, 1990)† and KADS:

- The Generic Task approach makes the underlying problem-solving method

† The description given in this article is much more conceptual and therefore better comparable to KADS than other publications.

explicit (e.g. goal decomposition). In KADS this is implicit in the task knowledge description.
- In the Generic Task approach only the method description is domain-independent: its application to a task-domain is, unlike KADS, described in domain-specific terms (Allemang, 1992). This limits the reusability of the resulting model.

In the computationally-oriented approaches, the underlying assumptions about the reasoning techniques supported by the approach tend to bias the problem-solving model. For example, in a pure constraint-satisfaction approach the idea of grouping will usually not be considered and will also not be easy to include.

WEAK POINTS OF THE APPROACH

In this application of KADS, some weak points that have already been pointed out (see Schreiber *et al.*, 1993: chapter 1), become very clear:

- If no interpretation model is available, the knowledge engineer has to construct a model almost from scratch.
- The typology of knowledge sources described in Breuker *et al.* (1987), and used quite rigorously in this article, does not always provide appropriate distinctions between inferences. For example, knowledge sources of type *select* appear in many places in the model presented and range from trivial selections to inferences involving complex knowledge structures (*select 2–4*).

Bob Wielinga and Werner Karbach provided extensive comments on earlier versions of this article.
The research reported here was largely carried out in the course of the KADS-II project, and was partially funded by contract P5248 (KADS-II) of the ESPRIT programme of the European Commission. The partners in the project are Cap Gemini Innovation (France), Cap Gemini Logic (Sweden), the Netherlands Energy Research Foundation ECN (The Netherlands), ERITEL (Spain), IBM France (France), Lloyd's Register (United Kingdom), the Swedish Institute of Computer Science (Sweden), Siemens AG (Germany), Touche Ross MC (United Kingdom), the University of Amsterdam (The Netherlands) and the Free University of Brussels (Belgium). This paper reflects the opinions of the author and not necessarily those of the consortium.

References

ALLEMANG, D. (1992). Modelling a configuration problem with Generic Tasks. In M LINSTER, Ed. *Sisyphus '91: models of problem solving*. GMD Report no. 663, St Augustin, Germany.

BREUKER, J. A., WIELINGA, B. J., VAN SOMEREN, M., DE HOOG, R., SCHREIBER, A. T., DE GREEF, P., BREDEWEG, B., WIELEMAKER, J., BILLAULT, J. P., DAVOODI, M. & HAYWARD, S. A. (1987). *Model driven knowledge acquisition: interpretation models.* ESPRIT Project P1098 Deliverable D1 (task A1), University of Amsterdam and STL Ltd.

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

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

ESHELMAN, L. (1988). MOLE: a knowledge-acquisition tool for cover-and-differentiate systems. In S. MARCUS, Ed. *Automating Knowledge Acquisition for Expert Systems*, pp. 37–80. Boston: Kluwer.

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

MUSEN, M. A. (1989). *Automated Generation of Model-Based Knowledge-Acquisition Tools.* Research Notes in Artificial Intelligence. London: Pitman.

PUPPE, F. (1990). *Problemlösungsmethoden in Expertensystemen.* Studienreihe Informatik. Berlin: Springer-Verlag.

SCHREIBER, A. T. (1992). *Pragmatics of the knowledge level.* PhD thesis, University of Amsterdam.

SCHREIBER, A. T., AKKERMANS, J. M. & WIELINGA, B. J. (1991). On problems with the knowledge level perspective. In L. STEELS & B. SMITH, Eds. *AISB-91: Artificial Intelligence and Simulation of Behaviour,* pp. 208–221. London: Springer-Verlag. Also in J. H. BOOSE & B. R. GAINES, Eds. *Proceedings Banff'90 Knowledge Acquisition Workshop,* pp. 30-1–30-14. University of Calgary: SRDG.

SCHREIBER, A. T., WIELINGA, B. J. & BREUKER, J. A., Eds (1993). *KADS: A Principled Approach to Knowledge-Based System Development.* London: Academic Press.

WIELEMAKER, J. (1991). *SWI-Prolog 1.5: Reference Manual.* University of Amsterdam, Social Science Informatics, Roetersstraat 15, 10-18 WB Amsterdam, The Netherlands. E-mail: jan@swi.psy.uva.nl.

WIELEMAKER, J. & BILLAULT, J. P. (1988). *A KADS analysis for configuration.* ESPRIT Project P1098, Deliverable E5.1 Uva-F5-PR-001, SWI, University of Amsterdam. Available from: University of Amsterdam, Social Science Informatics, Roetersstraat 15, 1018 WB, The Netherlands.

WIELINGA, B. J., SCHREIBER, A. T. & BREUKER, J. A. (1992). KADS: a modelling approach to knowledge engineering. *Knowledge Acquisition,* **4**(1), 5–53. Special issue "The KADS approach to knowledge engineering". Reprinted in B. BUCHANAN & D. WILKINS, Eds (1992). *Readings in Knowledge Acquisition and Learning,* pp. 92–116. San Mateo, CA: Morgan Kaufman.