Designing Knowledge Based Systems: The CommonKADS Design Model

John K.C. Kingston

AIAI-TR-219

Also appears in *Research and Development in Expert Systems XIV*,
Proceedings of Expert Systems '97, the annual conference of the British
Computer Society's Specialist Group on Expert Systems, Churchill College,
Cambridge, 15-17 December 1997.

Artificial Intelligence Applications Institute,
University of Edinburgh,
80 South Bridge,
Edinburgh, EH1 1HN
United Kingdom

Abstract

The problem of designing a knowledge based system well relies on the knowledge engineer's programming skills, and on his ability to devise, remember, and dynamically update a design specification. This is a difficult task for all but the smallest knowledge based systems.

These problems can be alleviated by producing representations of the expert's knowledge and of the design specification in the form of text or diagrams. The best known approach for producing such documents is the CommonKADS methodology, particularly its Expertise Model, which models expert problem solving. However, the Expertise Model is intended to represent knowledge at a level of abstraction which is independent of implementation; it neither allows representation of, nor gives guidance on, decisions about which programming techniques to use in order to represent the acquired knowledge. The responsibility for these activities is passed to the CommonKADS Design Model.

This paper describes the three-stage approach to KBS design recommended by the Design Model (choosing an overall approach to design, choosing ideal knowledge representation and programming techniques, and deciding how to implement the recommended techniques in the chosen software), as well as outlining possible sources of guidance for making good selections of knowledge representations and inference techniques. It then illustrates the use of the Design Model for two systems, one for machine fault diagnosis and one for mortgage application assessment. These systems have been developed by AIAI, and CommonKADS Expertise models for both these systems have been published in [Kingston1993].

1 Introduction

The problem of designing a knowledge based system well is one of the most frequent problems that knowledge engineers face. When knowledge based systems are developed by rapid prototyping, good design relies on the knowledge engineer's programming skills, and on his ability to devise, remember, and dynamically update a design specification. This is a difficult task for all but the smallest knowledge based systems, especially if the system intermixes expert knowledge with system control operations (MYCIN did this, which was a primary reason for the failure of the GUIDON system [Heycke1995]). It is possible for the system to get out of control so that even its author cannot understand why apparently small changes have large effects on the overall system.

These problems can be alleviated by producing representations of the expert's knowledge and of the design specification in the form of text or diagrams, thus documenting the expert's knowledge and the important design decisions independently of the system. The best known approach for producing such documents is the CommonKADS methodology [Breuker & van de Velde1994] [Schreiber *et al.*1994b] [Wielinga1993], which proposes several diagram-based models which reflect knowledge from different perspectives and at different levels of abstraction. The most widely used component of CommonKADS is the Expertise Model, which models expert problem solving in three components: domain (declarative) knowledge, inference (procedural) knowledge and task (control) knowledge. It also provides a couple of libraries of generic models to support re-use.

However, the Expertise Model is intended to represent knowledge at a level of abstraction which is independent of implementation; it neither allows representation of, nor gives guidance on, decisions about which programming techniques to use in order to represent the acquired knowledge. The responsibility for these activities, which are essential for modular design and efficient implementation, is passed to the CommonKADS Design Model. The Design Model was specified towards the end of the CommonKADS project [van de Velde & others1994]; apart from a worked example published by the project team [Schrooten1993], little or nothing has been published describing its use in realistic applications. The purpose of this paper is to describe the CommonKADS Design Model, including suggestions on diagrammatic representations of the model, and on sources of guidance for making design decisions. The paper illustrates the use of the Design Model by reverse engineering two existing KBS systems to show how the CommonKADS Design Model would have applied to them. The example systems are the same ones which were described in [Kingston1993]; the earlier paper includes expertise models of both systems.

2 The CommonKADS Design Model

The CommonKADS Design Model is intended to support knowledge engineers in choosing knowledge representations and programming techniques in order to produce a good design of a KBS system. It aims to do this in a way which is both generic (i.e. platform-independent for as long as possible, thus opening up possibilities for reusability) and economical (it encourages preservation of the structures within the expertise model). It also makes use of the CommonKADS Communication Model [Waern *et al.*1994] as a starting point for user interface design.

The Design Model supports selection of representations and techniques by encouraging the designer to start with the knowledge contained in an expertise model, and to perform a three-stage transformation process in order to produce design recommendations. These three stages are:

- *Application design*: choosing an overall approach to design decomposition.
- *Architectural design*: choosing ideal knowledge representation and programming techniques
- *Platform design*: deciding how to implement the recommended techniques in the chosen software.

2.1 Application Design

The application design is the first of these three stages. The purpose of application design is to decompose the knowledge into manageable "chunks". The size and content of each chunk depends on the approach to decomposition which is used. Broadly speaking, three approaches to decomposition are available:

- Functional decomposition
- Object-oriented decomposition
- AI paradigms

Functional decomposition involves treating each inference step from the Expertise Model as being a "chunk" of functionality. Functional decomposition is therefore a *structure-preserving* approach to design, because the form of the inference structure is maintained in the design specification. The benefits of this are that the KBS will replicate the expert's problem solving process (or whatever improved process was modelled); any inference step which is identified as a canonical inference (see [Aben1994]) will have some of its expected functionality already defined; and perhaps most important of all, preserving the inference structure usually preserves the task structure from the Expertise Model as well. The task structure is very important for KBS design because it provides a semi-formal specification of the required flow of control for knowledge based processing, while the Design Model only provides a high-level textual description. Knowledge engineers therefore need to use both the Design Model and the task structure as a specification for KBS implementation.

Object-oriented decomposition treats each concept from the domain model as being a "chunk" of data - i.e. each concept is treated as an object class. Since concepts have properties with values, and relationships with other concepts, it's often helpful to represent concepts as objects. Object-oriented decomposition preserves the structure of the domain models in the expertise model; indeed, CommonKADS domain modelling can be seen as a generalisation of object oriented data modelling [Jansweijer1996]. It is also possible that some of the inference and task structures may be retained, since the CommonKADS inference structure is closely related to the Object Management Technique's Functional model, while the task model can be compared with OMT's Dynamic model; however, this technique is likely to have difficulty in assigning production rules[1] because they refer to more than one object, whereas production rules can usually be assigned fairly easily to functional "chunks".

Another option for knowledge engineer is to decide that an "AI paradigm" – a well-known approach to AI problem solving – is appropriate. These AI paradigms might include blackboard systems, constraint-based programming, qualitative simulation or model-based reasoning. In this case, the "chunks" of knowledge may be constraints, knowledge sources, or whatever is appropriate for the chosen approach. If an AI paradigm is chosen, it may be that little of the structure of the expertise model will be maintained; in practice, this means that the knowledge engineer will either have identified the likelihood of an AI

[1]See [Schreiber *et al.*1994a] for a description of *expressions*, which is CommonKADS' technique for representing production rules

paradigm being appropriate earlier in the development process, and will have customised the expertise model accordingly, or AI paradigms will be considered unfavourably because of the extra effort required to re-analyse the knowledge. Exceptions to this rule would be the use of a blackboard architecture (where only the task structure of the Expertise Model needs to be revised) or the use of model-based simulation to perform diagnostic tests on a system, under the overall control of a diagnostic inference structure.

Once decomposition has been performed, it's necessary to characterise the contents of each "chunk" in a way that specifies further design requirements. For example, if functional decomposition has been performed, it's helpful to designate the operation being performed by each inference step in the form of an *architectural command* – a "function name" which describes the action which the function performs. Typical operations might be *subset*, *get-property-value*, or *calculate*. As mentioned above, the definitions of canonical inference steps in the CommonKADS expertise model may be helpful in defining appropriate architectural commands; for example, an inference step of type *select-subset* is very likely to be implemented by a *subset* command. This process also helps validate the Expertise Model; if the architectural command differs significantly from the inference step definition, then a possible error in labelling or understanding the inference structure has been highlighted. A full set of possible architectural commands has not been published, but a suggested BNF for these commands is given in [Schrooten1993].

2.2 Architectural Design

The task of architectural design is to define a computational infrastructure capable of implementing all the architecture commands defined in the application design. It is at this stage that the preferred knowledge representation and inference techniques are selected.

Knowledge representations available to knowledge engineers typically include objects, facts, and production rules, as well as more "conventional" representations such as tables or arrays. Many programming techniques are available including data- and goal-driven reasoning, truth maintenance, meta-rules, and various search strategies. The architectural commands specified during the previous phase provide guidance to the knowledge engineer on which representations and techniques are appropriate; for example, a *get-property-value* operation specifies a preference for objects as a knowledge representation technique. The emphasis in this phase is on choosing *ideal* techniques; the appropriateness of these for the available software should be considered in the next phase. In practice, most knowledge engineers know which tool they will be using when this phase is performed, and so will not select representations or techniques which will be impossible to implement; this phase is still useful, however, in assessing the appropriateness of the chosen tool or the chosen AI paradigm.

It is at this stage of design that the experience of a knowledge engineer can be brought to bear in making good design decisions. If the knowledge engineer knows that a particular technique or representation has proved suitable (or otherwise) for a similar problem in the past, then a knowledge engineer can use this information to guide his choices. There have been some attempts to capture and encode this knowledge for the use of less experienced knowledge engineers; it turns out that there are a large number of features of knowledge based problems which affect the choice of representations and techniques, so many that an entire book has been filled with *probing questions* [Kline & Dolins1989]. Probing questions ask if certain features are present in a knowledge-based problem, and suggest suitable functionality based on that feature. An example of a probing question is given below:

> `On average, do we know five or more new facts about a domain object simply by being told that it is of type X?`
>
> `OR`
>
> `Are these new facts not known with certainty, but assumed unless there is evidence to the contrary?`
>
> `Yes` → `Place the object in a data structure (e.g.` *frames, semantic nets* `or` *objects*`) whose` *inheritance mechanism* `will provide the facts when needed, and whose` *default values* `will be assumed unless an exception is specifically asserted.`
>
> `No` → `Assert the new facts explicitly, which is a 'cheap' solution.`

The book cited above contains probing questions based on successful AI systems up to the time of publication. There is a need for further development of probing questions to keep pace with new technologies and techniques; AIAI has done some internal knowledge acquisition and system development in this area (see [MacNee1992] or [Kingston1995]), but there is a strong need for further research and development of probing questions.

2.3 Platform Design

The final phase of the CommonKADS Design Model considers how (or whether) the ideal knowledge representations and inference techniques should be implemented in the chosen software. Most modern KBS tools support both objects and rules, so knowledge representation is rarely a problem. However, some programming techniques can be awkward to implement; for example, implementing data-driven reasoning in a tool which primarily supports backward chaining. The restrictions of the tool may mean that a different programming technique needs to be used.

3 Worked Example 1: IMPRESS

The use of the CommonKADS Design Model will be demonstrated with two worked examples – IMPRESS, which diagnoses faults in plastic moulding machinery, and X-MATE, which assesses the risk of mortgage applicants failing to make repayments. These two projects have been chosen because their expertise models have been described in some detail in a previous paper [Kingston1993]. The design models used in these projects have been reverse engineered, to show how the decisions which were actually taken during system design would have been represented if a CommonKADS design model had been developed.

IMPRESS (the Injection Moulding Process Expert System) diagnoses the causes of faults in plastic injection mouldings. Given data about the type of fault (e.g. "black specks in the moulding"), IMPRESS considers all possible causes of the fault, suggests tests for the system user (a technician or machine operator) to perform on the system, and iterates through a cycle of test-discard hypotheses-suggest tests until there is only one hypothesis left.

The inference structure for IMPRESS is shown in Figure 1.

3.1 IMPRESS: Application Design

No AI paradigms appeared to have overriding advantages for IMPRESS, so the choice of application design became a choice between functional and object-oriented decomposition. A few relations had been identified at the domain level, and a detailed inference structure with a little extra procedural ordering information had also been developed.

It was decided to break down the expertise model using functional decomposition. The chosen functions are described in Table 1 It can be seen from the architectural commands that IMPRESS requires a *subset* operation, where a set (of fault states) is reduced to a smaller set which are compatible with all observed symptoms and measurements; several *get-property-value* operations, which obtain values such as the expected value of an observable if a particular hypothesis is true; a *sort* of tests according to the time required to undertake them; a *transfer task* which initiates processing outside of the current program; in this case, it asks a user to perform a test (which will observe or measure some relevant parameter of the machine), and to report the measured value to IMPRESS; and a *match-2* operation (a match between 2 values) to compare an observed measurement against the expected value of that observable in each fault state.

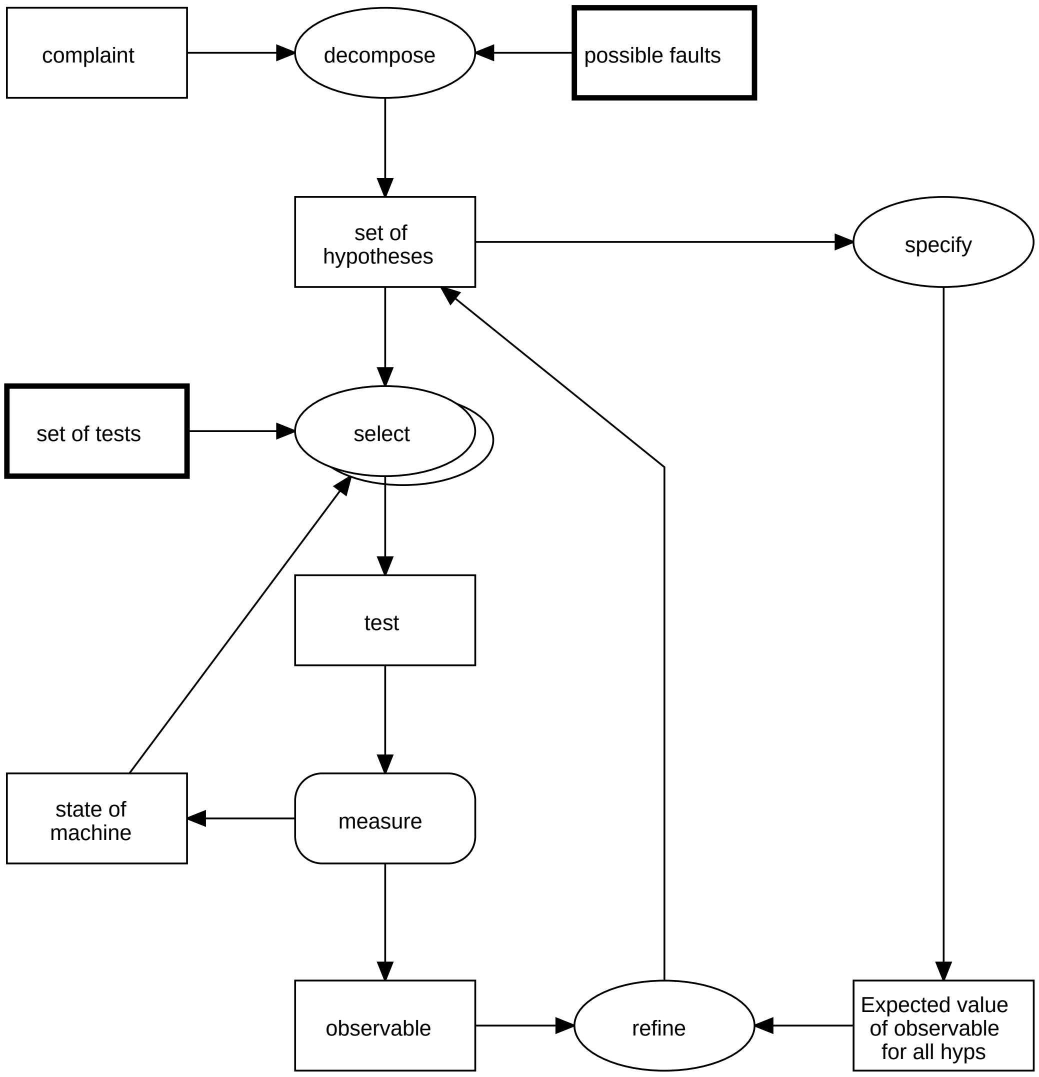

Figure 1: Inference structure for IMPRESS

Inference step	**Function**	**Arguments**
decompose	subset	:set all-faults :set hypotheses :key symptom
specify	get-property	:concept hypothesis :property expected-value :key observable
select	get-property	:concept hypothesised-fault :property distinguishing-observables
	subset	:set all-tests :set discriminating-tests :key distinguishing-observables
	get-property	:concept test :property time-required
	sort	:set discriminating-tests :key time-required
measure	Transfer Task	
refine	get-property	:concept hypothesised-fault :property expected-value :key test
	match	:element observed-value :element expected-value
	subset	:set hypotheses :set remaining-hypotheses :key difference

TABLE 1: Application Design for IMPRESS

An interesting observation on this mapping is that the *decompose* inference step in IMPRESS is mapped to a *subset* operation, whereas CommonKADS' definitions of canonical inference actions suggests that *decompose* requires replacing a single concept with a set of its component concepts. The reason for this difference is highlighted in [Breuker1997], where he points out that all tasks may have 3 types of solution: case data (the underlying cause of the problem), conclusion (an individual faulty item) or argumentation structures (the justification for the conclusion). The inference structure for IMPRESS was based on the generic inference structure for tasks requiring systematic diagnosis (see [Breuker & van de Velde1994] for the latest version of this), which presupposes that the required solution is a conclusion (a single faulty component). However, faults in plastic moulding machinery are rarely caused by a single faulty component, but rather by a combination of "components" (e.g. a coin stuck in the injection nozzle) or by inappropriate actions (e.g. running the machine at too high a temperature); the required solution is therefore a fault *state*, which corresponds to case data in Breuker's classification. The initial step in IMPRESS' diagnosis is therefore determining a relevant subset of all possible fault states, rather than identifying a set of machine components.

3.2 Architectural Design

The architectural design for IMPRESS' domain knowledge was relatively simple; fault states, tests and other concepts were implemented using objects, and domain relations were represented using slots. Set membership was also indicated using a slot, which carried the name of the set, and possible values of *Yes* and *No*.

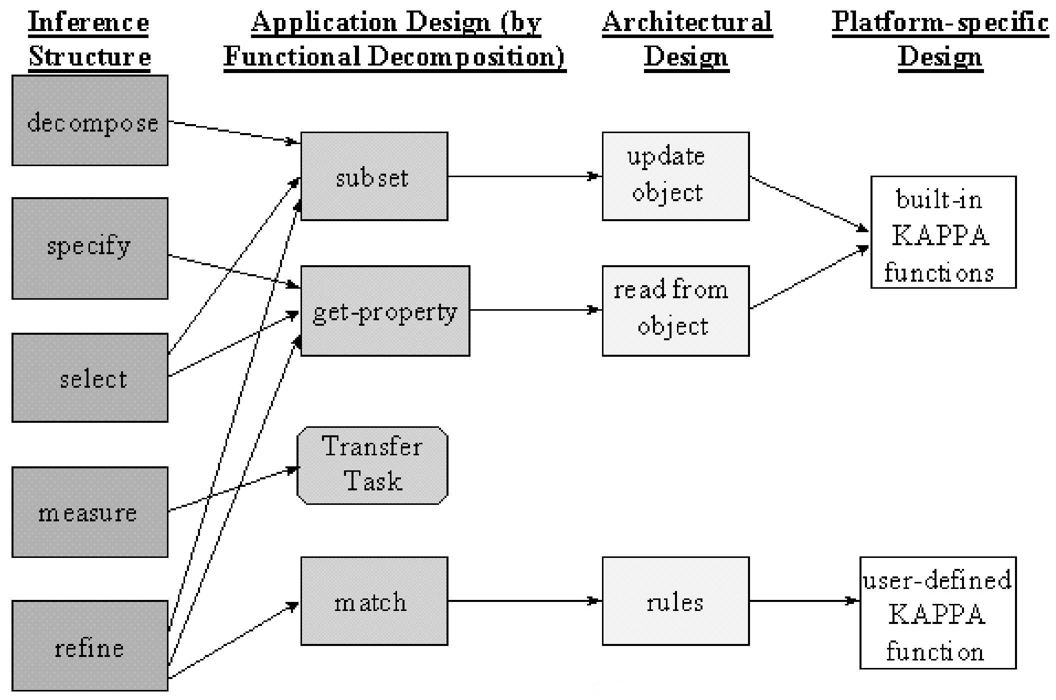

Figure 2: IMPRESS Design Model: Processes

The design for the inference steps identified a preference for production rules to carry out the *match* step. The other steps were identified as capable of being implemented with simple object-based operations: the *subset* operation involved changing values of the "set membership" slot from Yes to No, while the *get-property* operation requires reading the value of a slot in an object. The measurement task was considered to be a transfer task, so the only design requirements were for the user interface to instruct the user on the task, and obtain the result correctly; user interface design is considered elsewhere.

3.3 Platform Design

IMPRESS was implemented in KAPPA-PC on a Compaq 386. KAPPA-PC provided good support for object representations and object accessing functions, so the relevant architectural design recommendations were followed exactly. However, the rule system in that version of KAPPA-PC effectively operated as an add-on module to the rest of the system; it needed to be carefully set up and explicitly invoked. It was decided that, since the matching algorithm only needed to match 2 parameters (test results against faults), and there were approximately 40 faults and 40 tests in the knowledge base, then it was feasible to perform the matching with a doubly-iterative function, thus avoiding the need to introduce the rule system into the program at all.

The full design model for processes can be represented in a diagram (Figure 2).

3.4 Flow of Control

Design decisions on flow of control are made on the basis of the task structure from the Expertise Model. The knowledge representations and inference techniques recommended by the Design Model must be chained together in order to replicate the *task body* specified. For IMPRESS, the task body specifies a generate-and-test approach: an initial set of candidate faults is identified, and then the system enters a REPEAT-UNTIL loop in which tests are selected, performed, and the set of possible faults is narrowed down, until the set of faults has 1 or less members in it. This was easy to implement in KAPPA-PC.

3.5 IMPRESS: Design Modelling for Knowledge Representation and User Interfaces

The same process can be followed for making and recording decisions on knowledge representation design and user interface design. The starting point (i.e. the left-hand column of the Design Model) for knowledge representation design is the "knowledge roles" which appear in the inference structure of the Expertise Model; the starting point for user interface design is the inter-agent transactions which are identified as necessary in the Communication Model. The resulting diagrams for IMPRESS are shown in Figures 3 and 4.

4 Worked Example 2: X-MATE

X-MATE (EXpert Mortgage Arrears Threat Advisor) [Kingston17 18 Sep 1991] was developed for a large UK building society by Hewlett Packard's Knowledge Systems Centre with assistance from AIAI. Its task was to assess the likelihood of mortgage applicants meeting their loan repayments.

The building society's problem was that the percentage of defaulters was too high, and it was difficult to enforce quality control on acceptance of applications because, within certain guidelines, the acceptance or rejection of applications was almost entirely at the discretion of the local branch manager. The system was intended to support a branch manager or branch clerk by highlighting applications which were worthy of further investigation, and assisting the user in performing some further checks on the application. It did this by identifying the key features of "typical high risk customers", determining what data on the application form would indicate these features, and then scanning application forms (and, if necessary, data supplied from other sources) for the presence of these high risk indicators.

The inference structure for X-MATE is shown in Figure 5.

4.1 X-MATE: Application Design

X-MATE was also decomposed using functional decomposition. The application design for X-MATE can be seen in Table 2.

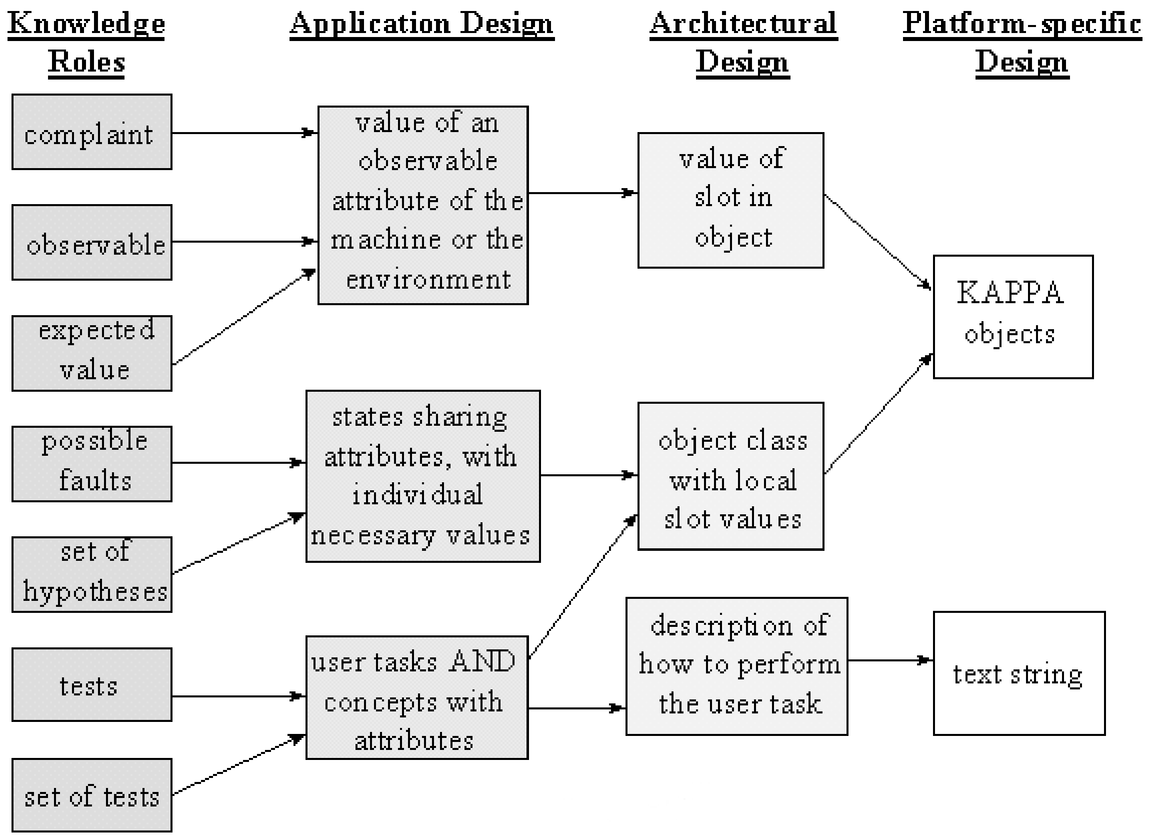
Knowledge Roles
Application Design
Architectural Design
Platform-specific Design
complaint
observable
expected value
possible faults
set of hypotheses
tests
set of tests
value of an observable attribute of the machine or the environment
states sharing attributes, with individual necessary values
user tasks AND concepts with attributes
value of slot in object
object class with local slot values
description of how to perform the user task
KAPPA objects
text string

Figure 3: IMPRESS Design Model: Domain Knowledge

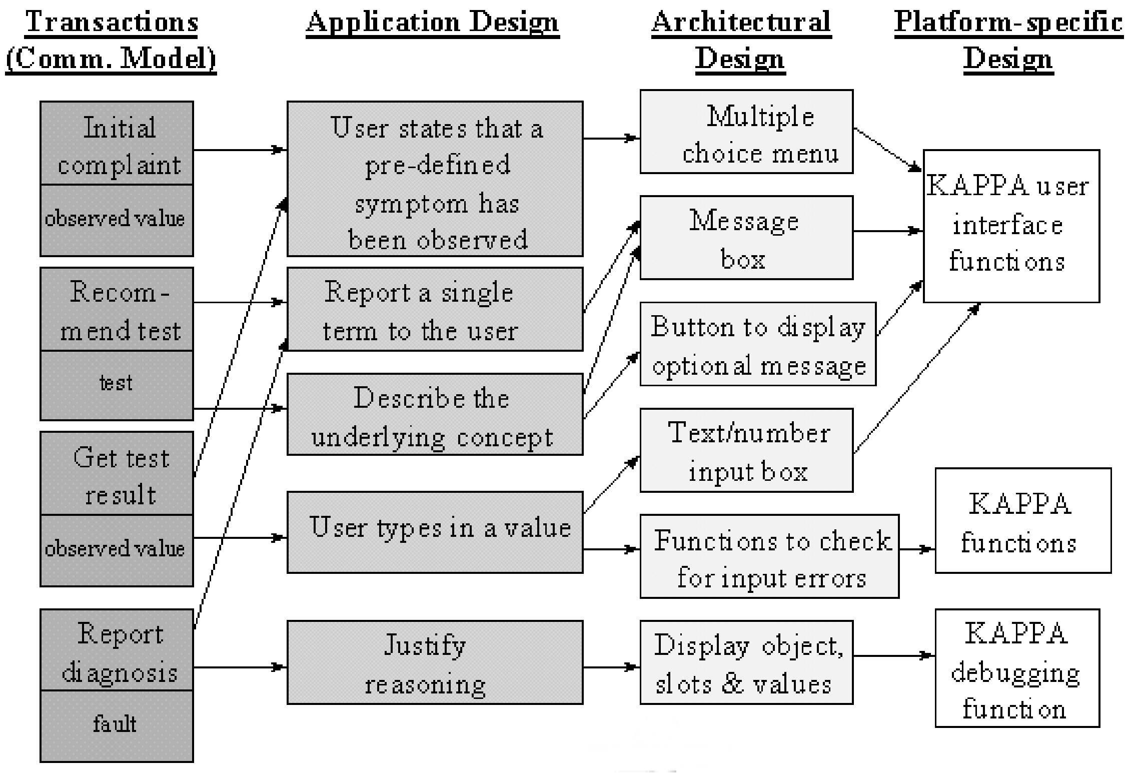

Figure 4: IMPRESS Design Model: User Interfaces

The most obvious factor about this design is that several inference steps are labelled "pre-compiled", and no architectural commands are defined for these steps. What has happened is that several of the problem-solving steps required to perform mortgage application assessment are considered to be have been carried out in advance by the experts who supplied the knowledge for the system; the system only contains their "distilled wisdom" on what to look for. In AI terminology, the "deep knowledge" of the full problem-solving process is replaced by "shallow knowledge" of associations between key inputs and important outputs.

The application design also contains an extra problem solving step (the selection of a particular data source) which did not appear in the inference structure. This extra step reflects a design decision to run the system up to four times, using different sets of data; the reason for this was to speed up processing by making all automatic checks first, and only proceeding to ask the user to input data if the application is deemed to be medium or high risk. It was therefore necessary to select the appropriate data source for each run.

The *select-simple* function is given a list of four data sources; its functionality is to select the next data source from the list. *match-N* performs pattern matching between a variable number of items, while *calculate* performs arithmetic calculations.

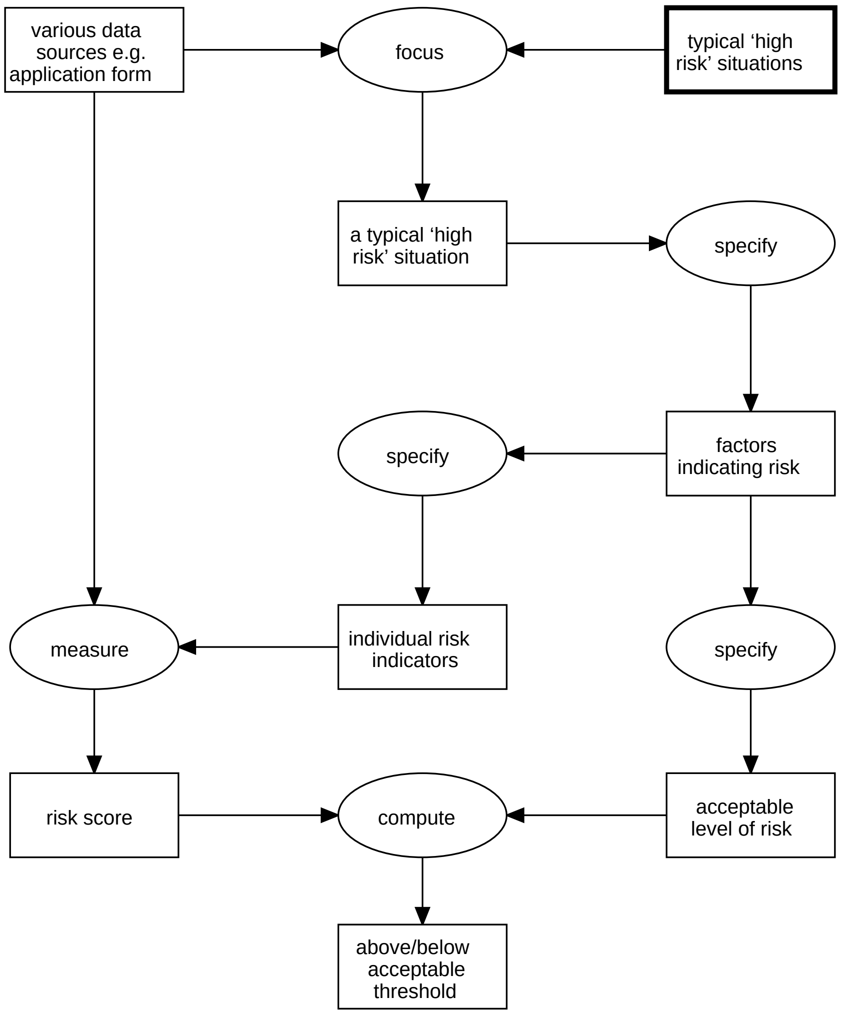
various data sources e.g. application form
focus
typical 'high risk' situations
a typical 'high risk' situation
specify
specify
factors indicating risk
measure
individual risk indicators
specify
risk score
compute
acceptable level of risk
above/below acceptable threshold

Figure 5: Inference structure for X-MATE

Inference step	Function	Arguments
focus	subset	:set all-risk-indicators :set some-risk-indicators :key situation
select	select-simple	:set available-data-sources :key phase-of-problem-solving
specify	pre-compiled	
specify	pre-compiled	
specify	pre-compiled	
measure	match-N	:elements application-form-data :elements risk-indicators
compute	calculate	:number risk-score :number risk-threshold

TABLE 2: Application Design for X-MATE

4.2 Architectural Design

The architectural design for X-MATE's processes is as follows:

- Select data sources: the key to this selection is the phase of processing. It can be implemented as a *case* statement i.e. "if phase 1, select source X; if phase 2, select source Y; etc."
- Matching should be implemented using production rules. Note that the recommendation for production rules is much stronger than it was for IMPRESS, because X-MATE correlates multiple factors in order to determine risk, whereas IMPRESS only matched 2 types of object. The theoretical set of possible matches is therefore much larger in X-MATE.
- Focus on a set of risk indicators: choose an appropriate rule set.
- Computation should be implemented using arithmetic functions.

The application form was represented using 2 or more objects: one object for each applicant (instances of a *Applicants* class) and one to represent the "case" (details of the property, and other non-applicant-specific information).

4.3 Platform Design

X-MATE was implemented in KAPPA-PC 1.1 on a HP Vectra 386 PC. The platform design mirrored the architectural design; no changes were deemed necessary.

The full design model for processes in X-MATE can be seen in Figure 6.

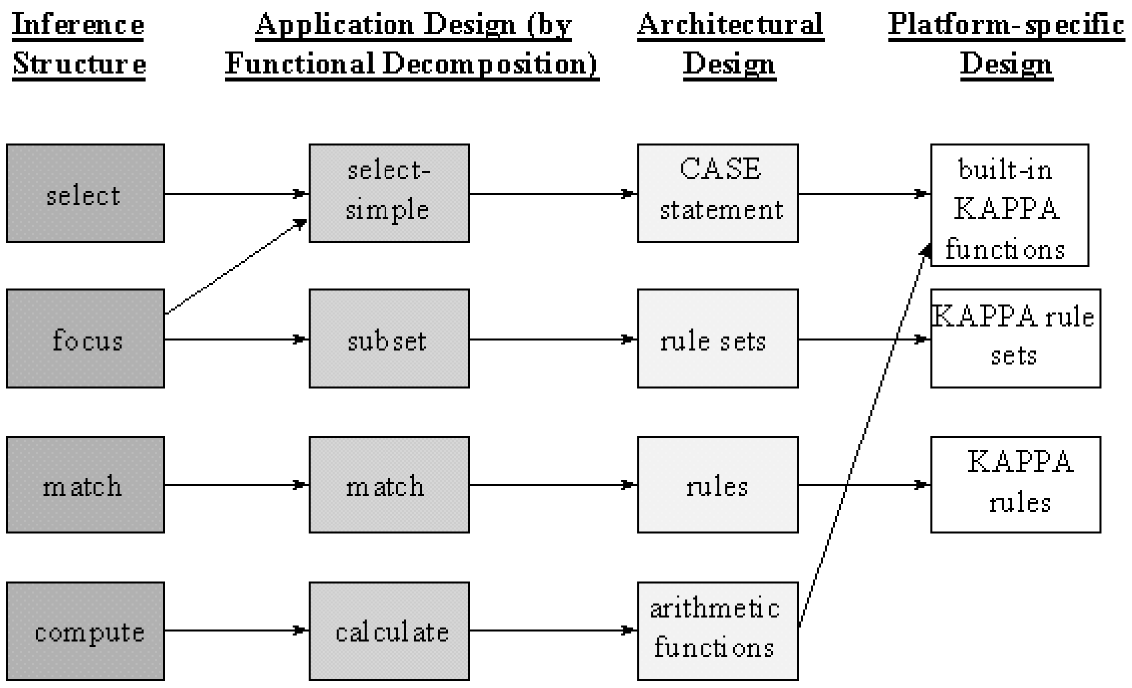

Figure 6: X-MATE Design Model: Processes

4.4 Flow of Control

The flow of control specified for X-MATE is to `repeat` running through the *whole* inference structure `until` the computed risk score doesn't meet a particular threshold, or until there are no more rule sets to be processed. When an application comes in, the first rule set is selected and is run on the objects representing the applicants and the case. If the resulting risk score does not reach a certain threshold, the application is deemed OK; if it does reach the threshold, another rule set is loaded and run on the same objects *after* extra attributes have been added by an automatic request to a credit search bureau. If a second threshold is breached, a third rule set is loaded which asks the user to draw out (mostly) textual data from the application and accompanying references; if another threshold is breached, then the system loads in its final ruleset, which requires further questions to be asked of the applicants themselves.

The final accumulated risk score is then recorded and can be displayed later, or sorted to produce a list of the riskiest applications for forwarding to Head Office. The system has been designed not to reject any applications without further consultation.

5 Conclusion

It can be seen that the CommonKADS Design Model is a useful way of recording design decisions, and of viewing how one design decision flows from another;

it therefore provides useful documentation of the process of system design. The separation of flow-of-control design from selection of representations & techniques is a consequence of a similar separation in the Expertise Model; this encourages greater modularity and reusability of designs. The three-stage design process helps to validate the Expertise Model and to separate decisions on good design techniques from decisions on what can be implemented.

Weaknesses in the Design Model include a lack of guidance on selection of techniques; probing questions provide some remedy for this. The lack of a defined set of architectural commands is also a weakness.

In summary, the CommonKADS Design Model is a useful aid to knowledge engineers in representing and recording design decisions, especially if an Expertise Model and a Communication Model have been developed previously. The usefulness of the Design Model will be improved by further recommendations on content (particularly architectural commands) and guidance on making selections (i.e. development of further "probing questions").

References

[Aben1994] Aben, M. 1994. *Formal methods in Knowledge Engineering.* Ph.D. Dissertation, SWI, University of Amsterdam. The relevant chapter is also available as CommonKADS report KADS-II/T1.2/WP/UvA/040/1.0.

[Breuker & van de Velde1994] Breuker, J., and van de Velde, W. 1994. *The CommonKADS Library: reusable components for artificial problem solving.* Amsterdam, Tokyo: IOS Press.

[Breuker1997] Breuker, J. 1997. Problems in indexing problem-solving methods. In Benjamins, R., ed., *Proceedings of the Workshop on Problem Solving Methods.* Nagoya, Japan: IJCAI-97.

[Heycke1995] Heycke, T. 1995. Historical projects. HTML document http://www-camis.stanford.edu/research/history.html, Center for Advanced Medical Informatics at Stanford.

[Jansweijer1996] Jansweijer, W. 1996. Recommendations to EuroKnowledge. KACTUS Deliverable KACTUS-DO1f.1-UvA-V0.2, University of Amsterdam.

[Kingston17 18 Sep 1991] Kingston, J. 17-18 Sep 1991. X-MATE: Creating an interpretation model for credit risk assessment. In *Expert Systems 91.* British Computer Society. Also available from AIAI as AIAI-TR-98.

[Kingston1993] Kingston, J. 1993. Re-engineering IMPRESS and X-MATE using CommonKADS. In *Research and Development in Expert Systems X*, 17–42. Cambridge University Press. http://www.aiai.ed.ac.uk/ jkk/publications.html.

[Kingston1995] Kingston, J. K. C. 1995. Applying KADS to KADS: knowledge based guidance for knowledge engineering. *Expert Systems* 12(1).

[Kline & Dolins1989] Kline, P. J., and Dolins, S. B. 1989. *Designing expert systems : a guide to selecting implementation techniques.* Wiley.

[MacNee1992] MacNee, C. 1992. PDQ: A knowledge-based system to help knowledge-based system designers to select knowledge representation and inference techniques. Master's thesis, Dept of Artificial Intelligence, University of Edinburgh.

[Schreiber *et al.*1994a] Schreiber, G.; Wielinga, B.; Akkermans, H.; and de Velde, W. V. 1994a. CML: The CommonKADS Conceptual Modelling Language. KADS-II project deliverable, University of Amsterdam and others.

[Schreiber *et al.*1994b] Schreiber, G.; Wielinga, B.; de Hoog, R.; Akkermans, H.; and van de Velde, W. 1994b. CommonKADS: A Comprehensive Methodology for KBS Development. *IEEE Expert* 28–37.

[Schrooten1993] Schrooten, R. 1993. Sabena flight schedule case: an example of a design model. CommonKADS Deliverable D.M.7 KADS-II/M7/VUB/RR/064/2.1, Vrije Universiteit Brussel. This report has been included in the CommonKADS 'Design Model and Process' report, the number of which is given below.

[van de Velde & others1994] van de Velde, W., et al. 1994. Design model and process. KADS-II/M7/VUB/RR/064/2.1, Vrije Universiteit Brussel.

[Waern *et al.*1994] Waern, A.; Höök, K.; Gustavsson, R.; and Holm, P. 1994. The Common KADS Communication Model. ESPRIT Project P5248 KADS-II KADS-II/M3/TR/SICS, Swedish Institute of Computer Science.

[Wielinga1993] Wielinga, B. 1993. Expertise Model: Model Definition Document. CommonKADS Project Report, University of Amsterdam. KADS-II/M2/UvA/026/2.0.