Design Model and Process

Walter Van de Velde [1]
(editor)
with contributions from
Cuno Duursma [1]
Guus Schreiber [2]
Peter Terpstra [2]
Ronald Schrooten [1]
Vassilios Golfinopoulos [1]
Olle Olsson [3]
Ulf Sundin [3]
Maria Gustavsson [4]

[1] *Vrije Universiteit Brussel*

[2] *Universiteit van Amsterdam*

[3] *Swedish Institute of Computer Science*

[4] *Cap Programator*

Document Id.	KADS-II/M7/VUB/RR/064/2.1
CEC Reference	Deliverable D.M.7, Version 2.1
Classification	Consortium
Distribution	Consortium
Date	October 4, 1994

KADS-II Consortium

This document is part of a research project partially funded by the ESPRIT Programme of the Commission of the European Communities as project number 5248. The partners in this project are Gap Gemini Innovation (F), Cap Programator (S), Netherlands Energy Research Foundation ECN (NL), ERITEL SA (ESP), IBM France (F), Lloyd's Register (UK), Swedish Institute of Computer Science (S), Siemens AG (D), Touche Ross MC (UK), University of Amsterdam (NL) and Free University of Brussels (B).

Cap Gemini Innovation
118, rue de Tocqueville
75682 Paris
France
Phone: +33 1 40 54 66 66 ; Fax: +33 1 42 67 41 39
Contactperson: M. Georges
E-mail: mari@capsogeti.fr

Cap Programator
Box 226
S-721 06 Västerås
Sweden
Phone: +46 21 13 72 65 ; Fax: +46 21 13 00 49
Contactperson: W. Menezes
E-mail: wm@industri.vs.cgl.se

Netherlands Energy Research Foundation ECN
Software Engineering & Research Department
P.O. Box 1, NL 1755 ZG Petten
The Netherlands
Phone: +31 22 46 44 58; Fax: +31 22 46 18 64
Contactperson: Dr. J.M. Akkermans
E-mail: akkermans@ecn.nl

ERITEL SA
P. de la Castellana 141
28046 Madrid
Spain
Phone: +34 1 348 12 23 ; Fax: +34 1 279 10 74
Contactperson: J. Guerra
E-mail:

IBM France
La Pompignane
BP1021
34006 MONTPELLIER
France
Phone: +33 67 34 60 20 ; Fax: +33 67 34 67 81
Contactperson: G. Rondot
E-mail:

Lloyd's Register
29, Wellesley Road
Croydon CRO 2AJ, Surrey
United Kingdom
Phone: +44 816 814 040 ; Fax: +44 818 816 814
Contactperson: R. Martil
E-mail: rob@aie.lreg.co.uk

Swedish Institute for Computer Science
KBS Laboratory
Box 1263, 164 28 Kista
Sweden
Phone: +46 87 52 15 20 ; Fax: 46 87 51 72 30
Contactperson: R. Gustavsson
E-mail: rune@sics.se

Siemens AG ZFE 1S INF 32
Otto Hahn Ring 6
D-8000 Muenchen 83
Germany
Phone: +49 89 636 40798 ; Fax: 49 89 636 441 50
Contactperson: C. Bauer
E-mail: cb@ztivax.uucp

Touche Ross Management Consultants
Knowledge Based Systems Centre
Hill House, 1, Little New Street
London EC4A 3TR, United Kingdom
Phone: +44 ; Fax: +44 71 583 8517
Contactperson: L. Land
E-mail: Lise_Land@trkbsc.uucp

University of Amsterdam
Department of Social Science Informatics
Roetersstraat 15, NL 1018 WB Amsterdam
The Netherlands
Phone: +31 20 525 67 89; Fax: +31 20 525 68 96
Contactperson: B.J. Wielinga
E-mail: wielinga@swi.psy.uva.nl

Free University of Brussels
Artificial Intelligence Laboratory
Pleinlaan 2, B-1050 Brussels
Belgium
Phone: +32 2 641 2965 ; Fax: +32 2 641 v2870
Contactperson: W. Van de Velde
E-mail: walter@arti.vub.ac.be

This document was produced using the LaTeX wordprocessor

Design Model and Process

Walter Van de Velde [1]
(editor)
with contributions from
Cuno Duursma [1]
Guus Schreiber [2]
Peter Terpstra [2]
Ronald Schrooten [1]
Vassilios Golfinopoulos [1]
Olle Olsson [3]
Ulf Sundin [3]
Maria Gustavsson [4]

[1] *Vrije Universiteit Brussel*
[2] *Universiteit van Amsterdam*
[3] *Swedish Institute of Computer Science*
[4] *Cap Programator*

Synopsis

This document summarizes results of Task M7 "Design Model". The material is re-arranged in several parts to form a comprehensive reference document on the *Common*KADS design model.
Part I of the document contains the design model definition, rationale, notations and risk factors, and also defines the framework for a design library. In Part II, the relationships between design model and other models is described, and the concept of structural design is introduced and exemplified. Structure-preserving design of the expertise model is described, and the concepts of and a tool for design configuration by constraint solving is presented. Part III contains a case study in the use of the design model. In one appendix an extensive illustration of the use of quality and risk in a design context.

Document Id.	KADS-II/M7/VUB/RR/064/2.1
CEC Reference	Deliverable D.M.7, Version 2.1
Classification	Consortium
Distribution	Consortium
Date	October 4, 1994

Revisions

Author: Walter Van de Velde
Revision date: June 20, 1993
Modifications: Update of 1.0 to 1.1 after formal review and plenary. Comments from Santiago Gala, Winifred Menezes and TM.7 partners incorporated. Minor changes in introduction. Model template described more extensively with respect to the relations. One reference to agent model added in platform design.

Author: Walter Van de Velde
Revision date: June 24, 1993
Modifications: Clarified architecture notion and its relation to application and platform. Introduction of states expanded. Format for states added and exemplified. To-do part on modeling guidelines added at the end. Clarification of distinction between two types of guidelines.

Author: Olle Olsson
Revision date: December 17, 1993
Modifications: Generation of new draft version. Integrated Design Notation chapter (chapter from DM7.2a, Peter Terpstra). Integrated appendix on Quality Assurance for BlackBoard Systems (Golfinoupolous and Van de Velde, VUB). Integrated appendix containing the Sabena-case design model (chapter from DM7.2b, R. Schrooten). Integrated chapter on Dependencies with Other CommonKADS Models (Olle Olsson, SICS, and Maria Gustavsson, CP)). Integrated chapter on Structural Design (Olle Olsson, SICS). Integrated chapter on Library Framework (Olle Olsson, SICS). Modifications to the format of guidelines.

Author: Olle Olsson
Revision date: December 27, 1993
Modifications: Generation of version 2.0, based on input from: Peter Terpstra (UvA), Roland LeMeur (CGInn), Maria Gustavsson (CP), Winifred Menezes (CP).

Author: Olle Olsson
Revision date: May, 1994
Modifications: Generation of version 2.1. Added a chapter on compositional design (Duursma and Schrooten, VUB). Added a chapter on structure-preserving design for the problem solver (Schreiber, UvA). Added appendix containing complete example corresponding to this chapter on

structure-preserving design. Moving Sabena case study from appendix to separate section. Clarifications added to chapter on library framework. Reviewers: Steve Welles (LR, Guus Schreiber (UvA).

Preface

This document summarizes results of Task M7 "Design Model". It addresses a gap in the KADS-I methodology, namely the step from a conceptual model to a running KBS. A comprehensive methodology for KBS development can not ignore this issue.

The area of software engineering is well researched and text books on the engineering of Artificial Intelligence software are beginning to appear. The problem is, however, difficult and unresolved, and the so called "software crisis" is still with us. As our contribution we could have chosen to develop only experimental tools and techniques for automatic programming, transformational implementation or inductive programming. Though the application of AI techniques to the programming problem itself holds much promise, an exclusive focus on this approach would fail to satisfy urgent needs of practical software engineering in KBS projects. A valuable contribution to the *Common*Kads methodology is to make contact with what is going on in day to day software engineering.

What we propose in this document are the *Common*Kads specific elements of KBS design. Our opinion is that no new *Common*Kads design methodology can realistically replace or integrate with existing approaches, neither that any of the existing approaches is best for *Common*Kads. We rather view our work as contributing KBS specific elements to three interrelated approaches to design: conventional software engineering, structured design methods, and formal design methods. Our strategy has been to single out what are the powerful ideas in the *Common*Kads methodology and to see what they can bring to solving the problem of designing high-quality software. These relevant ideas are the model and state perspective, structure preserving design and the use of a formal modelling language.

This document partly elaborates on material that was previously available in the document "Operationalising Knowledge Models" [Duursma et al., 1992]. The completed material is being re-arranged in several parts that will evolve to a comprehensive reference document on the *Common*KADS design model. The present version contains in Part I complete descriptions of the design model definition, rationale, notations, risk factors, and a framework for a design library. Part II describes how the a design depends on the contents of other *Common*Kads models, and how the structures of these other models can direct the design process. In appendices larger case studies are presented, illustrating the design model, and how design decisions relate to quality and risk factors. Subsequent versions will elaborate more on the modelling (i.e., design) approaches and the library contents.

Contents

Chapter 1

Introduction

Knowledge systems are programs and programs need to be implemented. A complete methodology for the development of knowledge systems therefore needs to tie into the broader practice of software engineering. Software development is not only a costly process. As software is being applied in critical, sensitive or strategic situations, its quality is becoming more and more important. Software must be bug free, usable, understandable, correct, maintainable, portable and so on and so forth (see e.g. ISO 9001 [9001, 1987] and [Williams et al., 1992]). To reach these objectives with as few resources as possible requires a principled and disciplined approach.

This document presents elements of a *Common*Kads approach toward the design of knowledge systems. This may seem overly ambitious. Software engineering, even of conventional[1] information systems, is indeed a vast field on its own and still struggling to meet its own objectives. After years of research and experience the task of developing and maintaining complex software remains a difficult one. For knowledge systems we shouldn't expect the problems to be easier. Indeed, we content, as does Derek Partridge in a recent book on engineering AI software:

> "...,AI is not really a special case: it is merely an extreme, and as such it serves the very useful purpose of highlighting problems that run throughout the software spectrum, as well as presenting us with a challenge in its own right" [Partridge, 1992, p199].

How is software engineering for knowledge based systems extreme? Without pretending to be exhaustive, here are some of these characteristics:

- Application Systems: knowledge based systems are applications in the basic sense of the word. They are applied directly to concrete problems in concrete situations and directly by users active in a domain of activity. They require no plan or procedure to apply them (as opposed to, for example, a statistics software package) but immediatly tie into the user's dayly practice. This means that they must be grounded in real-world domains and highly integrated in the day to day activity of

[1] We use words such as "conventional" and "traditional" to qualify software engineering techniques and methodologies that are being practiced today in non-KBS projects. These terms are by no means intended to minimize the quality or importance of these approaches.

the people using them. This implies also that they must withstand to be (mis-)used in unexpected ways.

- Competence Oriented Approach: knowledge based technology is based on competence models rather than on performance models. The algorithmic aspect of a knowledge system is secondary to the knowledge content contained in the collection of knowledge items that together determine the competence of the system. The detail of the behavior induced by this competence (i.e., by the close to rational application of the knowledge items by the computational architecture, mostly based on controlled search for a consistent model) is not the primary objective of system development.

- Ill-Specified Tasks: knowledge based systems deal with ill-specified tasks in the sense that algorithmic solutions brake down on them because of epistemological or pragmatic limitations [Steels, 1990]. Rather domain and situation specific knowledge is needed to overcome the limitations of applying and using knowledge. This requires extreme flexibility for developing and maintaining systems, essentially because solutions to deal with epistemological and pragmatical problems are highly context and situation dependent.

- Strong Interaction between Knowledge Elements: the knowledge embedded in a system determines its competence, to the extend that the bits and pieces can be found and exploited in the right situation and context. Extreme flexibility in knowledge use entails a problem of knowledge interaction, resulting in unpredictable problem solving behaviors. The problem is agravated by the fact that knowledge keeps on evolving which puts high demands on incremental refinement and maintenance.

These characteristics have required original technical solutions. Search has traditionally been a topic of research in AI and techniques for implementing search using various control regimes have been developed. Constraint satisfaction is another example of techniques that are being used in AI programming. New software architecture have been developed to implement complex programs in AI. The metaphor of cooperating experts working on a common blackboard has lead to the development of the blackboard architecture. Rule based systems are being used with various degrees of sophistication. Truth and consistency maintenance systems have become a standard tool in the AI programmer's toolkit.

Software specification techniques have not been given much attention in AI. AI software simply was not specified, rather developed in an incremental and problem-driven fashion. The target of solving ill-defined tasks made it quite impossible to specify in advance what a well-defined solution could look like. However, as knowledge engineering is coming to grips with the difficult issues involved in knowledge based problem solving, the requirements for defining dedicated problem solvers is becoming better understood. Simple functional specifications are not sufficient for this purpose, but more advanced tools are available - even if their usage is not yet common practice.

On the other hand user-requirements are problematic and their importance has been long underestimated. Success of a knowledge based system will depend crucially on the way in which it ties into the daily work practice of users. What exactly these users want or need and how this can be best integrated in their environment is a difficult issue which has not been studied systematically.

Rapid prototyping is the paradigmatic example of software engineering practice in AI. It is an extreme case of an approach different from the traditional waterfall model of software development. It has long been thought that AI programming is simply incompatible with classical programming until it was realized that waterfall models are equally paradigmatic and illusive. Also traditional software development is a complex process.

What can one then hope to contribute in such a difficult and well-researched field such as software engineering? Should one develop a new knowledge systems specific methodology for software design? Should one take up some of the existing approaches and integrate them within *Common*Kads? Or should one try to provide general solutions, maybe based on advanced AI technology such as automatic program synthesis? The following section outlines the approach that we have taken.

1.1 Towards a *Common*Kads Design Approach

In the ESPRIT project KADS (P1098) a KADS approach to KBS design was put forward. In this approach the conceptual model (model of expertise and model of cooperation) and external requirements were taken as the input to the design process. This process elaborates three viewpoints on the artifact under construction: the functional viewpoint, the behavioral viewpoint and the structural viewpoint. The issues that were addressed by that work are summarized below [Schreiber et al., 1987] [Schreiber et al., 1989] [Schreiber et al., 1990]:

- Functional viewpoint
  - Can we define a paradigm for KBS design within KADS?
  - What type of functional blocks are there within such a paradigm?
- Behavioral viewpoint
  - Can the reasoning methods used in Artificial Intelligence be classified according to some criterion and put in a library of methods?
  - How can the entries from the library of methods be linked to types of functional blocks?
  - What type of design elements are there?
  - How are design elements related to the meta-classes and/or domain knowledge in the analysis model and how can we support the transformation of the latter in the former?
- Structural viewpoint
  - What environments are there
  - What methods and corresponding design elements do environments support

The results of these initial investigations in P1098 are to set up a general framework, to indicate elements of a library of paradigms, methods and skeletal architectures. The idea of structure preserving design was described as "the logical thing to do", while at the same time the potential role of formal models was recognized.

The present work can be viewed as a natural continuation of the lines of thought set out in KADS-I. However the work is scoped as follows:

1. focus on KBS specific design issues;
2. clarify the link to conventional software engineering methodology;
3. use the modelling perspective as the way to capture the what of design;
4. capture process in risk-driven design guidelines;
5. explore potential of both structured and formal design approaches;
6. provide library support for design and implementation

Note that we have chosen not to explore the tempting research avenue of applying AI techniques to the software engineering problem itself. This would have led into automatic programming, inductive programming or transformational techniques. Surely more interesting and spectacular than down to earth software engineering but also that much further away from day to day practice.

1.2 Purpose and overview of this document

In this document the design model is described according to the general framework for describing the models in the CommonKADS model set. A complete treatment of the design model covers at least four aspects [de Hoog et al., 1993b]:

Rationale: the rationale for the model justifies its being part of the CommonKADS model set in general terms. It illustrates the usefulness of the model and its independence of the other model. Apart from the general rational a particular instantiation of a model has a project specific rational, e.g., development of a preliminary design model for feasibility study. This is documented by the knowledge engineer.

Template: the model template describes the specified structure of the model, i.e., the entities, slots and relations that describe its contents. The template specifies the possible elements of a filled in model but a specific project may choose to instantiate the model only partially or may choose to add objects or slots. The template provides a strong and useful guidance.

States: the model states capture useful intermediate development stages of a model, described in terms of structural and quality features of the model under development. They are described in terms of one or more situations in which it is useful to pursue that state, and also mention the justification in terms of product or project qualities that are under threat. States are the basic guidance in risk driven model development.

Approaches: approaches focus on the overall process of model development. Approaches are described in terms of activities and techniques, and justified in terms of their relation to a quality model. They are most easily described in terms of paradigmatic scenarios.

This document completely covers the first three points. As illustration of the design model, an example has been included in this document. An approach to design is presented, and pointers are given to eleborate descriptions of this approach.

The CommonKADS design notation is defined and motivated. The illustration of the design model mentioned earlier uses this notation. Another extensive and complete design case study is part of another document.[2]

Another aspect of a comprehensive study of a particular model is library support. It has been decided to develop a partial library of generic design models and design model components. The framework for this library is presented in this document. A later version will contain example library components.

A worked-out example of how to relate design model decomposition to a quality decomposition is given in an appendix. This illustrates concrete design guidelines for the case of blackboard architectures, and shows how design decisions can be guided by quality aspects.

[2] Peter Terpstra & Ronald Schrooten "CommonKADS specific design decisions and their notation" (KADS-II/M7/UvA/043/1.1). That document is the original source of both the chapter on design notation and the chapter containing a case study on design model use in this document.

Part I

Design Model Definition

Chapter 2

The Design Model

Walter Van de Velde, Cuno Duursma, Guus Schreiber

This chapter presents the details of the CommonKADS design model.

2.1 What is a Design

The word "design" itself is used in various meanings. In its broadest sense it covers the process of transforming an "informal idea" into a program description that can be readily implemented to operationalize the idea. In this meaning it involves much more than just coming up with the software. For example the construction of detailed requirements or, for the case of knowledge systems, the decisions on the conceptual level would be considered part of the design. A recent handbook [Kline and Dolins, 1989] considers as part of design the choice between, for example, causal or heuristic reasoning. In *Common*Kads these choices are reflected in modelling decisions in the expertise model and out of the scope of the design model.

*Common*Kads is model oriented [Wielinga et al., 1992]:

> *The development of a knowledge-based system is seen as the construction of a set of models of problem-solving behavior, seen in its concrete organizational and application context. A KBS is a computational realization associated with these models.*

Figure 2.1 (taken from [Wielinga et al., 1992]) summarizes the suite of models involved in the *Common*Kads methodology (see [de Hoog et al., 1993b] for a comprehensive description of the models and the dependencies between them). The design model describes the computational realization of the expertise and communication model within the boundary conditions specified by organization, task and agent model. More precisely it is a model of a problem solving behavior in computational and representational terms.

Within the modelling view the following is our definition of design:

> *In* Common*Kads we consider as design the process of creating a software design that fits the constraints and requirements that are implicitly or explicitly*

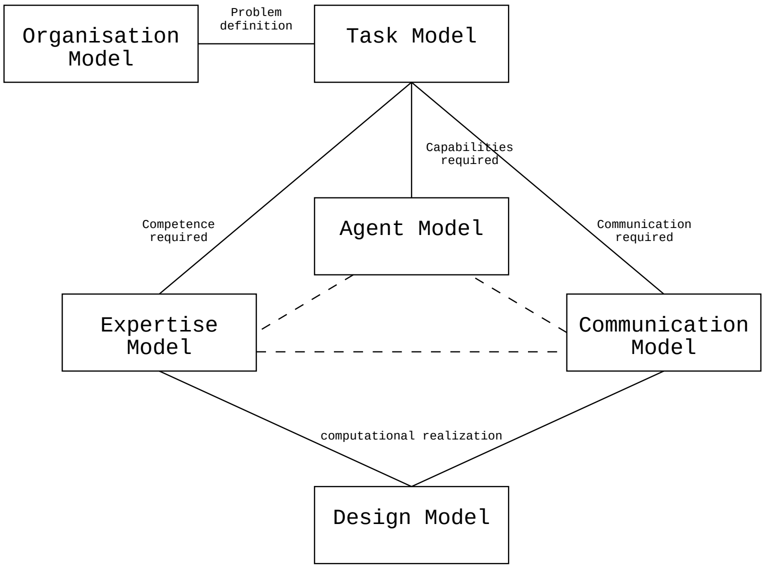


Figure 2.1: The position of the design model in the *Common*Kads suite of models. The organization model is the tool to analyse an organization. The task model captures the global task within the organization that deals with a certain function that needs support, including the assignment of tasks to agents. The agent models describe agent capabilities, the expertise model agent competence involved in realizing the overall task. The communication model describes the communication behavior among the agents. The design model describes the structure and mechanisms of the artifacts - usually knowledge based systems - involved in the task. The lines indicate direct dependencies between elements of the models.

> *contained in the relevant pieces of the* Common*Kads product model, i.e., the organizational models, the task models, the expertise models, the agent models and the communication models.*

In its product view, this modelling view is not incompatible with "mainstream" software engineering:

> "A general model of a software design is a directed graph where the nodes represent entities in the design such as processes, functions or types and the links represent relations between these design entities. The target of the design process is the creation of such a graph where there are no inconsistencies and where all the relationships between design entities are legal ones" [Sommerville, 1989, p178]

This "general model" needs to be better specified and refined to be practical.[1] Those specializations reflect paradigms for design. They can be distinguished according to principle

[1]It is another question whether software engineering uses the term model in the same sense as does *Common*Kads. In *Common*Kads the design model must be viewed as a model of (problem solving) behavior, whereas software engineering seems to refer more to a model of the software. In its purest view the design model is a model of a KBS solution to a problem in an organization. This solution is not just a KBS, but a whole process within the organization for which the KBS is an enabling element. Other elements may be needed as well, for example paper forms or measuring tools. When the design model is

of decomposition. Object-oriented and functional design are two such paradigms. The *Common*Kads design model is a KBS specific refinement of this general model.

Taking a process perspective matters are less clear. Often software engineers create a design in an ad hoc fashion. Their purpose is to build a program that "does the job" and to reach that end all means are allowed including, sometimes, changing the definition of "the job". *Structured methods* are a first step toward a more principled design process. A structured method is basically a set of notations and guidelines about how to create a software design. They provide a means to express the design as a coherent set of elements of the design. Structured methods are better than no method at all, but have been criticised for being informal and incomplete. *Formal methods* go a step further. They not only provide for a precise way of describing a design but more importantly, they advocate a transformation-based design process: formal specifications are transformed into a more and more precise software description. In its most ambitious form - transformational programming - software is developed, modified and maintained at the specification level and then transformed into high-quality code [Green et al., 1986]. Whether formal methods are cost-effective in practice is an unresolved issue. For instance work toward formal requirements and specification engineering is not yet mature. Nevertheless the approach is attracting more and more attention from research and user communities alike [Lowry, 1992]. Both structured and formal design methods have their value. It is likely that they will continue to be used side by side, depending on project characteristics that, admittedly, are still to be discovered. A prescriptive view on the software development process is, however, illusive in both approaches. The *Common*Kads risk-driven process view can perfectly accommodate this.

2.2 Role of the Design Model

The *Common*Kads product model records at all times the outcomes of the important decisions that must be taken during a KBS project. In this sense it is an *output* of a project that can be viewed as the modeling equivalent of all the documents that one traditionally finds in software projects. Another view on the product model is to see it as a *resource* for driving the project. The risk driven approach that underlies the *Common*Kads life-cycle continuously requires the evaluation of the project state. The product model is used as the blackboard that records that state and it is elaborated as more and more information about the project is required to anticipate identified risks. This is why the product model need not be developed in equal depth in all its components.

This section focuses on the first of these roles for the design model. The second role, the role of resource for driving the development process, is treated in Section 3.2. The following are some of the most obvious roles of an explicit design component in the *Common*Kads product model. Of course, a knowledge based system's project will almost invariably have such a design component in the product model, though here again the model can be elaborated to various degrees of detail in its different parts.[2]

viewed as a model of the process in design terms then there is in principle a place for the design of all of these as well. This is clearly not a practically feasible scope but it is good to keep in mind that the design model as we have developed it is just the for us most relevant part of a more general product model. More general aspects of fielding a KBS are described in the organization model.

[2]The *Common*Kads model set can be instantiated to model situations that do not involve any knowledge

Separating analyses from implementation The design model is part of the CommonKADS model suite because it is good practice to separate analyses from actual implementation. It allows one to use the most appropriate language for analyses (e.g., domain models) and implementation (e.g., rules) and serves as a bridge between the two.

Quality assurance The traditional advantages of the separation between analyses and implementation remain valid, and are re-enforced by the complexity of some KBS projects. A design model serves as documentation for maintenance and quality assurance, as a means for assessment of feasibility and implementation cost, and as a guide for organizing the programming of the system.

Platform independent system specification The design model serves as a specification of a system, independent of the platform on which it is to be implemented. A CommonKADS project that does not deliver an implemented KBS can still deliver a precise design to the client.

Decomposing the design task The structure of the design model prescribes what needs to be designed, i.e., what are the major decisions to be taken by the designer. These decisions are, for example, how to decompose the application, what architecture to use, and what target language to choose.

Separating architecture from contents The gap between the conceptual analyses and the implementation is particularly wide for knowledge based systems. Standard programming languages or low level architectures are not of a sufficiently abstract level to easily map onto the concepts used for analyses (e.g., a model of expertise). This is the reason for the increasing interest in designing new architectures: the traditional ones simply are not good enough. The design model documents the computational architecture of the KBS.

Adding requirements from environment The design model takes into account requirements from the environment which are not and indeed can not be taken into account in the other models. For example, the model of expertise makes abstraction of computational mechanisms, the choice of which nevertheless influences for example run time or explanation requirements.

Re-usability of components Design models can serve as high level descriptions of re-usable components. For example, ready made architectures can be selected based on their design model.

Design of interface and interaction The scope of the design model, though originally intended to cover only the design of the KBS is extended to cover the design of any physical entity involved in realizing the KBS solution. This may include forms used in communication, interfaces, additional tools. The design model also accommodate design of conventional software.

system at all, for instance when an existing organizational situation is being analyzed.

2.3 Aspects and scope of the design model

The CommonKADS design can be decomposed along two dimensions. The first one distinguishes between *application design*, *architecture design* and *platform design*, and entails a transition from conceptual to implementational emphasis. The second dimension distinguishes types of subsystems: KBS, conventional systems and interaction subsystems. To each of these subsystems the distinction between application, architecture and platform level applies.

- Application design operates at the conceptual level of 'real-world' entities and tasks that have to be realized by the artifact.
- The architecture design refers to the abstract computational structures within which the application design is realized.
- The platform design specifies the target implementation language and the hardware and software platform that will be used.

In software engineering a distinction is made between software specification or high-level design, and detailed design [Sommerville, 1989]. The specification corresponds most closely to the *Common*Kads application design. Detailed design traditionally means the realization of the specification in a given computational environment. The detailed design is a detailed specification of code that can be easily programmed in the target language and environment. It corresponds to the *Common*Kads detailed architecture design and detailed application design. The dependencies between specifications and detailed designs are illustrated in Figure 2.2.

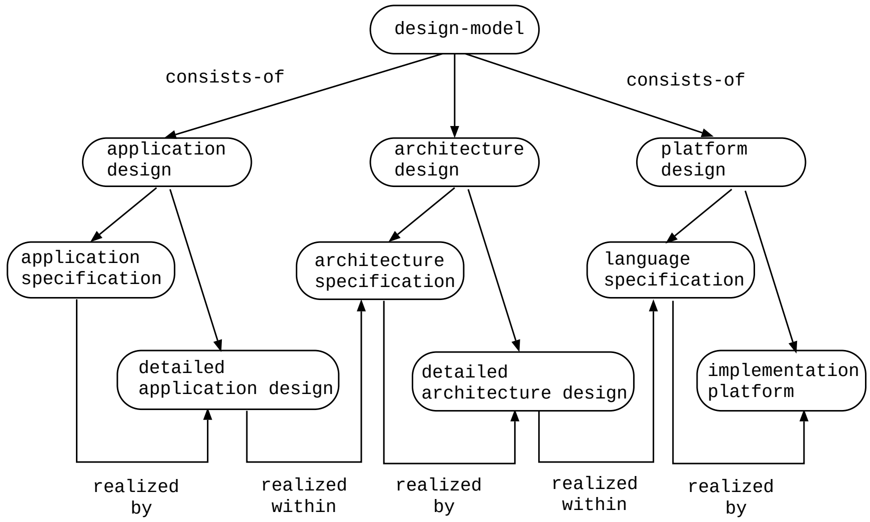


Figure 2.2: The specifications are realized by a detailed design. The detailed designs are program descriptions within the architecture and language that is specified.

We introduce an architecture design as a mediator between application design and platform design. Whereas application design is at the conceptual level, the architecture design is at an abstract computational level. An architecture could be viewed as just another implementation environment, but in knowledge based and AI applications it is often a highly sophisticated and dedicated piece of software. The role of the computational architecture is to bridge the gap between the implementation level and the conceptual entities that one finds in the application design. Traditionally simple concept and operations can be easily mapped to abstract data types in the target language. For more sophisticated applications this mapping becomes a mapping to an explicitly designed abstract machine that provides coarse grained computational building blocks. Typical examples of architectures are blackboard systems or task specific architectures. Typically an architecture also implements powerful reasoning primitives such as rule chaining or knowledge-source application.

Like traditional software engineering we also use the distinction between *specification* and *detailed design* at the level of application and architecture. Detailed design realizes the *means-ends* relation between an architecture and the application on the one hand, and between a target language and the architecture on the other. Detailed design uses a program definition language (PDL) to describe the realization of the specification (the ends) within a given computational environment (the means).

We have chosen to include only one additional means-ends relation in the design model. In principle an arbitrary number of such layers could be used. It has become common practice in software engineering to develop systems in a layered fashion, where each layer is built on top of "virtual machine" provided by the next layer down. Any of these layers has its own computational primitives and there can be as many layers as one likes or finds useful. A simple example would be an application built in an object-oriented environment, realized in prolog, prolog built in the Warren Abstract Machine, which is built on top of C, which is built on top of some assembler, down to the lowest level of the workstation.

Why then do we use only two levels? It seems we are missing lots of the design. There are two arguments for this decision. Firstly practice shows that the lower level means-ends relations do not involve KBS-specific design decisions. The detailed design of a diagnostic application in an object oriented fashion may involve KBS specific techniques, and so may the implementation in Prolog of certain sophisticated inheritance or delegation mechanisms in the object oriented architecture itself. But the implementation details of the Prolog do not matter to the application. Of course, the details of the Prolog language itself may matter, and these, as will be seen, can be modelled.

Another way to look at this is that the crucial step in the means-ends reduction is from application to computation. The application design is to be realized in an appropriate computational architecture. Below that architecture it is reduction from computation to computation, i.e., from one virtual machine to another. This is what computer science is good at and we should not bother about it here. If one wants to document those design aspects then one is free to do so, for example in the detailed design of the architecture. The structure of the model, however, does not explicitly support this and the designer can fall back on traditional design languages and methods.

In summary then, we choose to interpret the notion of architecture as a description of the computational entities on which the conceptual entities from the application design are being mapped. This we consider the crucial and most specific step of the design. In addition, since the architecture must match the conceptual level as close as possible (i.e., make the reduction as easy as possible) and must support often sophisticated inference techniques its specification may be non-trivial and its implementation may involve non standard designs. However lower level reductions are considered in the realm of computer science and left implicit, in the notion of target language.

The CommonKADS design model does not cover all aspects of design along the two dimensions mentioned before. Figure 2.3 illustrates the scope of the model. KBS stands for the knowledge-based sub-system, Interaction for the communication sub-system, and Other for other kinds of sub-systems. The rationale is to de-emphasize aspects that are not KBS-specific and do not obviously interfere with KBS-specific issues.

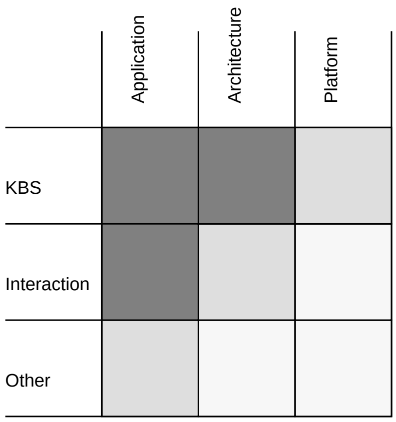


Figure 2.3: The scope of the design model focuses mainly on KBS specific issues. Black areas are extensively supported, grey areas less, while dotted areas are not explicitly addressed by the CommonKADS methodology.

2.4 The design model template

A brief note on the status and meaning of the design model template is in order. A generic design model can never be complete. Not only can we identify in advance all the design decisions that may be generally relevant. In addition some design decision impose themselves only through others. For example the choice of a blackboard architecture will imply a choice between central or distributed control over the knowledge sources. Thus, the design model (and probably any model in the CommonKADS model set) provides strong and hopefully useful guidance on what are the major *classes* of design decisions that need to be taken. At lower levels the individual decisions become to specific to be of any use in the general design model template.

A high-level decomposition of the design model is shown in Figure 2.4. We distinguish between three major classes of design decisions, namely related to *application design*, to *architecture design* and to *platform design*. In the following subsections we describe each of these in more detail. We treat them as three separate *constituents* of the design model.[3]

[3]The notion of constituent is similar to the one in the organization model.

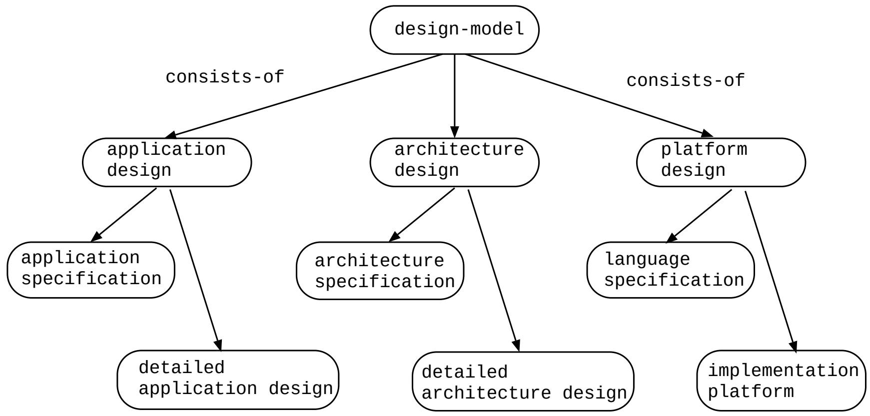


Figure 2.4: The top-level decomposition of the design model.

2.4.1 Application design

The application design documents basically the specification of the application except for the entity detailed application design. The application design is illustrated in figure 2.5.

Application The application specification consists of a high-level decomposition of the artifact in *sub-systems* and the conceptual decomposition of each of these sub-systems. There can be any number of sub-systems of any type.[4]

Application architecture *slot (Obl.) single*: [5]
The application architecture mentions the sub-systems and describes the top-level control between the sub-systems in terms of the functionality that is specified for each of the sub-systems.[6]

Has-subsystem *relation (Obl.) 1 to many*: The relation between an application and its sub-systems. Target is of type "sub-system" in the design model.

Sub-system The sub-systems of an application are the starting point for further specification. It mentions the type of sub-system and its functionality. Optional sub-system slots allow for hierarchical composition of sub-systems.

Name *slot (Obl.) single*: the name of the sub-system. Must be unique within the application.

Type *slot (Obl.) single*: The type of a sub-system. It is either *KBS*, *interaction* or another type, defined by the knowledge engineer. Thus sub-system can be a knowledge based system (type: KBS), an interaction sub-system (type:interaction) or a sub-system of conventional type (type, e.g., a data-base).

[4]Sub-systems can be but need not be hierarchically organized. This must be modeled by optional has-subsystem links between the various subsystems.

[5]*Obl* refers to a slot/relation that has to be instantiated for an identified object, *Opt.* refers to optional slots/relations, *single, many, multiple* refers to the number of allowed instantiations

[6]Application architecture is not to be confused with computational architecture in the *Common*Kads design model.

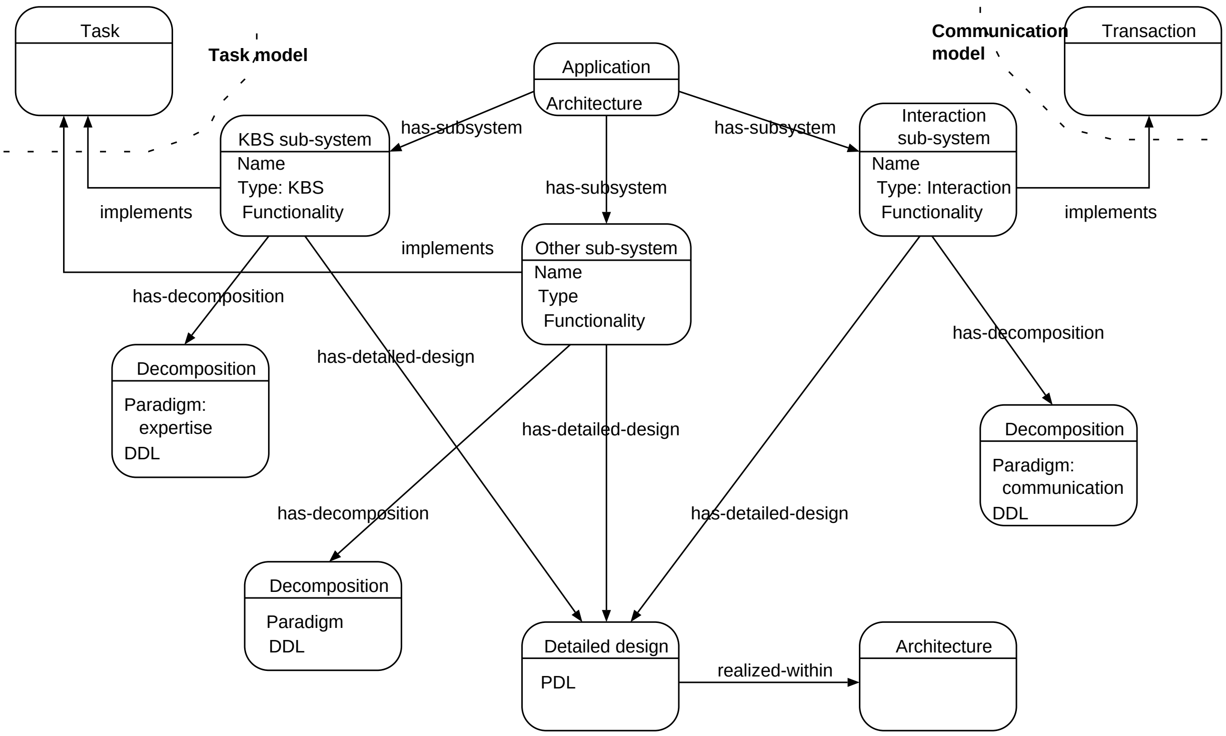


Figure 2.5: The application design

The different types of sub-systems are introduced because they require different design paradigms supported by specific bodies of design knowledge.

Functionality *slot (Obl.) multiple*: The functionality of a sub-system. Each sub-system implements a series of functionalities that are publicly available (i.e., serves as external interface) and can be used to specify top-level control in the application architecture.

Implements *relation (Obl.) 1 to many*: The relation between a sub-system and the task that it implements. Target is of type "task" in the task model.

Has-decomposition *relation (Obl.) 1 to 1*: The relation between a sub-system and its decomposition. Target is of type "decomposition" in the design model.

Has-detailed-design *relation (Opt.) 1 to 1*: The relation between a sub-system and its detailed design. Target is of type "detailed application design" in the design model. The relation is optional for sub-systems for type "other sub-system" because this will often be an existing one.

Decomposition Every sub-system has a *decomposition* into conceptual entities that realize the functionality.

Paradigm *slot (Obl.) single*: The paradigm of the decomposition.

The decomposition paradigm depends on the type of sub-system:

- KBS sub-system: The decomposition paradigm for KBS sub-systems is "expertise". The specification of a KBS sub-system thus closely follows the structure of the expertise model for the task(s) that the sub-system implements. Thus, the KBS specification is a design-specific elaboration of an expertise model. [7]

- Interaction sub-system: The decomposition paradigm for interaction sub-systems is "communication". The specification of an interaction sub-system thus closely follows the structure of the transactions (see communication model) that the sub-system implements.
- Other sub-system: The decomposition paradigm for other sub-systems can be anything. Two well-known paradigms from software engineering are function-oriented (or functional) decomposition and object-oriented decomposition. For a data-base on the other hand the conceptual schema (an entity relationship or NIAM diagram) could be used. One could also use a degenerate version of the CommonKADS decomposition paradigms for expertise and communication. For example a data-base could be decomposed according to an expertise paradigm with empty task layer, queries at the inference layer and data at the domain layer.

Note that the usage of expertise and communication model for software specification is an instance of what is called *model-based specification* in software engineering [Sommerville, 1989]. In *Common*Kads we argue that the expertise and communication models can be used as specifications but we do not intend to say that the specification and expertise or communication models are one and the same thing. Rather the specification is a design-specific elaboration of these models.

DDL *slot (Obl.) single* The design description language used to describe the elements of the decomposition. CommonKADS provides a specific design description languages for the decomposition paradigms of expertise and communication.

Detailed application design The sub-system specification is realized by a detailed design. The detailed design realizes the means-ends relation between a sub-system specification (the ends) and a computational architecture (the means).

PDL *slot (Obl.)* **single:** The program description language (PDL) that is being used for the detailed design. The PDL is derived from the architecture specification in the architecture design. The detailed design is then the specification of a "program" that can be run within that architecture.

Realized-within *relation (Obl.) 1 to 1*: The relation between the detailed application design and the architecture in which this design is realized. Target is of type "architecture" in the design model.

Instantiates *relation (Opt.) 1 to many*: The relation between the detailed application design and the computational objects or interface media in the architecture that it instantiates. Target is of type "computational object" or "interface

[7] It is in principle possible that a sub-system implement more than one task with different expertise models.

medium" in the design model. This can be used to document the architecture features that are actually used for a design.

Calls *relation (Opt.) 1 to many*: The relation between the detailed application design and the computational methods or interface activities in the architecture that it uses. Target is of type "computational method or "interface activity" in the design model. This can be used to document the architecture features that are actually used for a design.

The detailed application design bridges the gap between the conceptual level of application design and the computational level of the architecture. The closer the architecture maps to the type of objects that are being used in the application the easier this mapping will be. For example an object oriented architecture will easily support the detailed application design when the application specification has been done in a object-oriented fashion. Clearly the selection of an architecture that suites an application will be crucial for the ease of the design process.

2.4.2 Architecture design

Whereas application design focuses on the conceptual aspects of the design, architecture design focuses on the high-level computational infrastructure within which the artifact will be realized. The architecture design model consists of two parts: *architecture specification* and *detailed architecture design*. The architecture design model is shown in Figure 2.6.

Architecture specification The architecture specification describes the abstract machinery that is used to implement the functionalities of application sub-systems.

The design model covers two types of architectures: computational architecture and interface architecture.[8]

Computational architecture specification The abstract machinery that is used to implement the reasoning capabilities of an application. It consists of computational objects and computational methods to describe an architecture within a certain paradigm. The commands specification describes the external interface to the architecture.

Paradigm *slot (Opt.) single* The paradigmatic class of computational architecture, e.g., blackboard, rule-based. An architecture paradigm is a class of architectures that can be characterized by the computational objects and methods that it typically consists of. The details of these objects and methods may vary over different instances of the same paradigm. These are the additional design decisions needed to design the architecture.

Command *slot (Obl.) multiple* A specification of a command to the architecture that is externally available. An architecture typically has multiple commands. A method is specified in a model oriented way.

[8]This covers the KBS specific decisions.

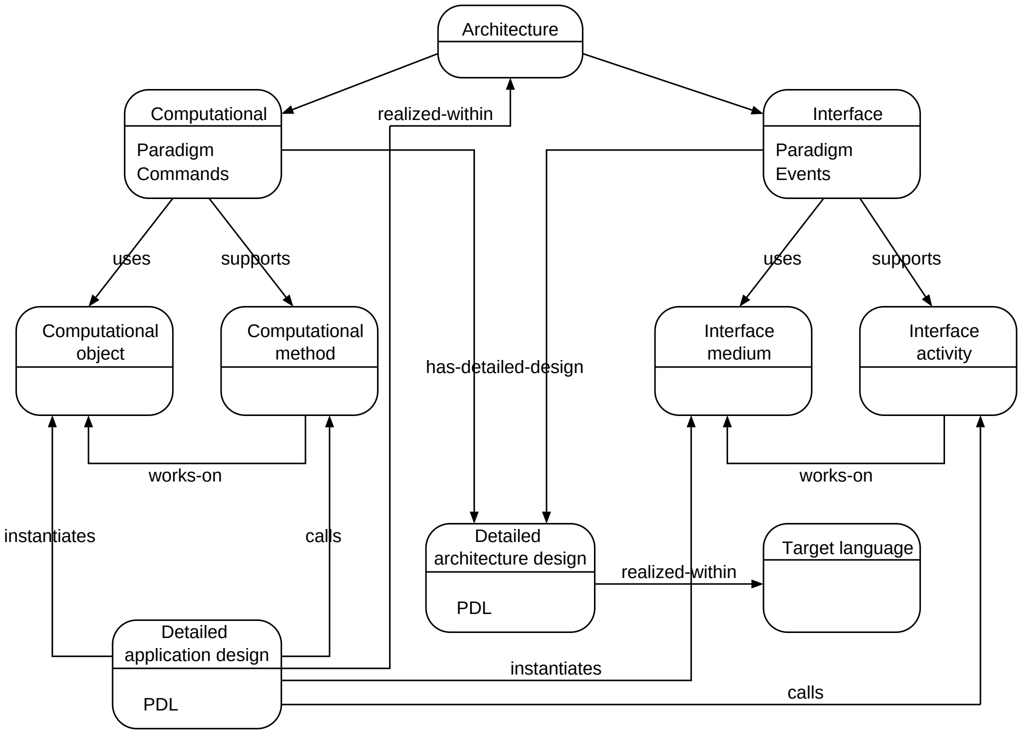


Figure 2.6: The architecture specification and its relation to detailed application and architecture design.

Uses *relation (Obl.) 1 to many* The relation between an architecture and a computational object that it uses. The target is of type "computational object" in the design model.

Supports *relation (Obl.) 1 to many* The relation between an architecture and the computational methods that it supports over the computational objects. The target is of type "computational method" in the design model.

Has-detailed-design *relation (Opt.) 1 to 1*: The relation between an architecture and its detailed design. The target is of type "detailed design" in the design model. This relation is optional since the architecture may be off-the-shelf and its implementation in the target language not relevant.

Computational object Computational objects constitute the computational ontology of the architecture. Here the basic computational entities of the architecture are specified. The specification of computational objects is like the specification of abstract data types.

Computational method Computational methods specify the effects of the basic mechanisms in the architecture that act over computational objects. Computational methods allow no external control. Methods are specified as elementary state transitions over the computational objects of the architecture.

Works-on *relation (Obl.) many to many*: the relation between the computational method and the computational object on which it operates. The target is of type "computational object" in the design model.

An architectural paradigm prescribes what the computational building blocks are, from which the artifact will be assembled, and what relations exists between them. An architectural paradigm can also prescribe how the analysis input should be mapped onto the architecture, or it can be tightly linked to an analysis approach. Two well-known architectural paradigms in software-engineering are functional architectures and the object-oriented architectures . In the functional approach the prime architectural components are functions. Functions are linked to each other via data flows. In the object-oriented approach the main components are objects. Objects are related to each other via associations (relations) and message connections. These architectural paradigms fit two well-known analysis approaches in software engineering, namely the functional approach [Yourdon, 1989b] and the object-oriented approach [Coad and Yourdon, 1991, Rumbaugh et al., 1991].

Another architectural paradigm that has been proposed for certain AI programs is the so-called "blackboard" architecture. In the latter paradigm the emphasis lies on the distribution of control in the system. This type of architecture has not yet a well established analysis method associated with it.

The three example architectural paradigms described above symbolize what appear to be three fundamental perspectives that one can take when describing a system [Rumbaugh et al., 1991, Yourdon, 1989a] (see also [Schreiber, 1992]), namely:

- the *data perspective*,
- the *functional perspective*, and
- the *control perspective*.

These three perspectives can be summarized as the "what, how, and when" views of a system. Choosing one of the paradigms does not mean that only this particular type of information is present in the architecture. For example, in the functional approach the data manipulated by functions must be described as well. Also, in the object-oriented approach operations (functions) and messages (control) must be defined for each object. The object-oriented approach in fact groups data, functions and control together in small units. The main distinguishing factor between the approaches is the *decomposition principle* that is employed when realizing a system. In the functional approach the functional perspective provides the decomposition principle: the system is decomposed into a hierarchical structure of functions and sub-functions. In the object-oriented approach the data perspective provides the entry point: system decomposition starts with building hierarchies of data objects.Many of the debates in software engineering have been about the "right" decomposition principle. More recent proposals in software engineering advocate a combination of all of them [Rumbaugh et al., 1991].

Interface architecture specification the abstract machinery that is used to implement the interaction facilities of an application. It consists of interface media and interface activities to describe an interface architecture within a certain paradigm. The events specification describes the external interface to the architecture.

Paradigm *Opt.* The paradigmatic type of interface architecture, e.g., line, window-based.

Event *Obl.* A specification of the events that cause interface activity. An interface architecture typically has many events.

Uses *relation (Obl.) 1 to many* The relation between an interface architecture and an interface medium that it uses. The target is of type "interface medium" in the design model.

Supports *relation (Obl.) 1 to many* The relation between an interface architecture and the interface activity that it supports over the interface medium. The target is of type "interface activity" in the design model.

Has-detailed-design *relation (Opt.) 1 to 1*: The relation between an interface architecture and its detailed design. The target is of type "detailed design" in the design model. This relation is optional since the interface architecture may be off-the-shelf and its implementation in the target language not relevant.

Interface medium A specification of the media that are used to interact with the user (or another system). Examples of media are windows, multi and hyper-media, ASCII line printing.

Interface activity The basic operations that the interface supports, for example, accepting a command, displaying some information. Other examples of activities are printing, clicking a button, and so forth.

Works-on *relation (Obl.) many to many*: the relation between the interface activity and the interface medium on which it operates. The target is of type "interface medium" in the design model.

The architecture specification provides one with the basic computational infrastructure that can be used in detailed application design (see previous section). The program description language (PDL) for detailed application design is derived from the architecture specification.

The architecture is implemented in the target language (see platform design) through the detailed architecture design.

Detailed architecture design The architecture specification is realized by a detailed design. The detailed design realizes the means-ends relation between an architecture specification (the ends) and a target language (the means).

PDL *Obl.*: The program description language (PDL) that is being used for the detailed architecture design. The PDL is derived from the target language specification in the platform design (or is even identical to it). The detailed architecture design is then the specification of a "program" that is intended for that language. It can be passed to a programmer team for straightforward implementation.

Realized-within *relation (Obl.) 1 to 1*: The relation between the detailed architecture design and the architecture in which this design is realized. Target is of type "target language" in the design model.

The detailed design of the interface architecture realizes the interface functionality within a target (specification) language (for example MOTIF or X11) that is being decided upon in the platform design. The detailed design of an interaction sub-system in the application instantiates the media and calls interface activities. Requirements for the interface architecture will come from the agent and communication model.

Note that the program description language for detailed application design and the one for detailed architecture design are different. The point of an architecture is to bridge the gap between the computational and the conceptual. Also note that references to other models can impossibly be specified as there is no relation at all between the chosen computational architecture and the more conceptual entities one finds in the expertise and communication model. As argued before it makes things a lot easier if such a mapping is explicitly aimed at when choosing an architecture and the design paradigm for the application specification.

2.4.3 Platform design

In platform design the lower level decisions about target language, hardware and software configuration, user and user environment are documented.[9] These can be different for any of the sub-systems in the application. The platform design captures aspects closest to the physical realization of the CommonKADS solution: where is the system installed, who is using it, what hardware and software configuration is required, and so forth. The platform design constituent of the design model is illustrated in Figure 2.7.

The main components of platform design are:

Platform *Obl.*: The platform design "hook".

- **Uses language** *relation (Obl.) 1 to many*: the relation between the platform and the target languages that it supports. The target is of type "target language" in the design model.
- **Required hardware** *relation (Obl.) 1 to many*: the relation between the platform and the hardware required for it. The target is of type "hardware configuration" in the design model.
- **Required software** *relation (Obl.) 1 to many*: the relation between the platform and the software required for it. The target is of type "software configuration" in the design model.
- **Has user** *relation (Obl.) 1 to many*: the relation between the platform and the users of the installed system. The target is of type "agent" in the agent model.
- **Has system model** *relation (Obl.) 1 to 1*: the relation between the platform and the model of the system as an agent. The target is of type "agent" in the agent model.

Target language *Obl.* A specification of a target language for implementation of an architecture. There can be many target languages, for example a different one for the architectures of the different sub-systems, but at least one. A PDL for the target

[9] The term "Platform" is used in a broader sense than just the computational platform. This may cause some confusion.

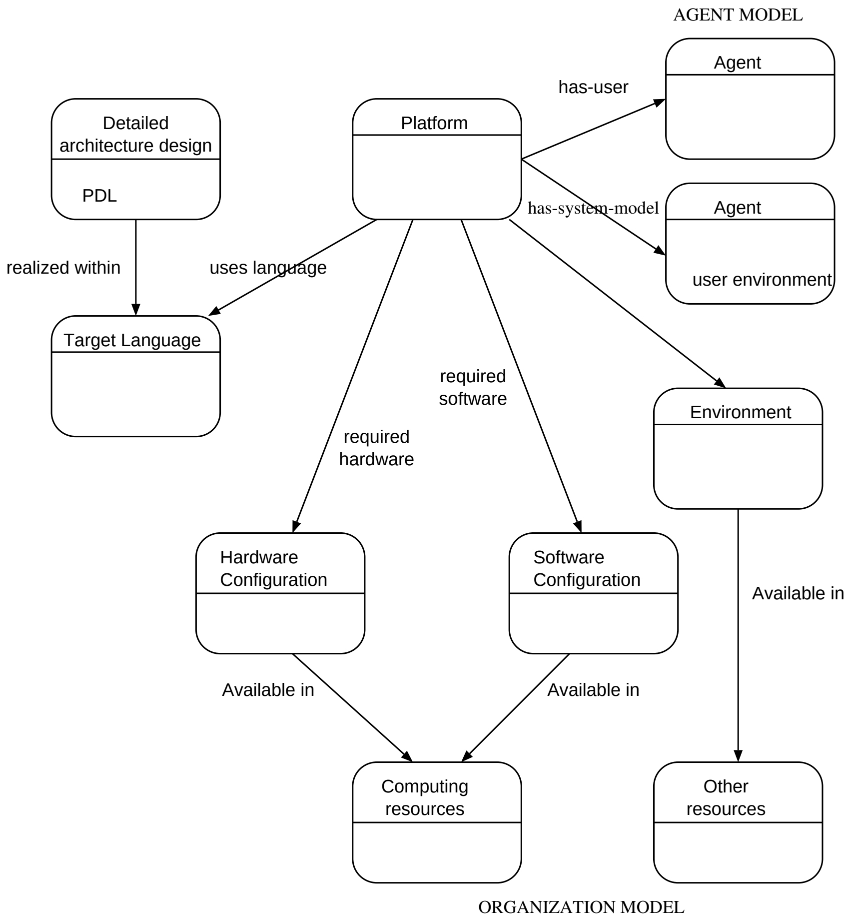


Figure 2.7: The platform design model.

language is used in the detailed architecture design for describing the program that implements the architecture in the target language.

User *Obl.* A reference to an agent (Agent model) that acts as a user of the system. There can be more than one user, but at least one.[10]

Hardware configuration *Opt.* A description of the hardware configuration referring to types of machines, connections, integration devices and so forth. Only those that are relevant to the system are mentioned. They should be available from the physical computing resources described in the corresponding constituent of the organization model.

Available in *relation (Opt.)*: The relation between the hardware configuration and the computing resources that is or must be available in the organisation. The target is of type "computing resource" from the (target) organizational model.

[10] Embedded KBS, i.e., those that themselves interact in real-time with their domain have no user. The specifics of such a KBS are not addressed in CommonKADS.

Software configuration *Opt.* A description of the (commercial) software that is needed, its configuration and versions. Only those that are relevant to the system are mentioned. They should be available from the logical computing resources described in the corresponding constituent of the organization model.

> **Available in** *relation (Opt.)*: The relation between the software configuration and the computing resources that is or must be available in the organisation. The target is of type "computing resource" from the (target) organizational model.

Environment *Opt.* A description of the work environment in which the system is to be installed. The environment should be available within the organization and is described in the "Other resources" constituent of the organizational model.

> **Available in** *relation (Opt.)*: The relation between the environment and the organizational situation that it is or must be available in. The target is of type "other resource" from the organizational model.

2.4.4 Summary of references to other models

In *Common*Kads we consider as design the process of creating a software design that fits the constraints and requirements that are implicitly or explicitly contained in the relevant pieces of the *Common*Kads product model, i.e., the organizational models, the task models, the expertise models, the agent models and the communication models. The design model documents all the important decisions that need to be taken about the design. Clearly then, the design model has numerous implicit or explicit dependencies to other models. In this section we summarize the explicit references of the design model to other models. This also indicates the basic consistency checks that can be performed on the product model.

KBS sub-system implements Task (reference to Task Model):

> A KBS sub-system is the actual realization of one or several tasks that have been assigned to an application. The Functionality slot of the KBS sub-system will reflect the Description of the Task.
>
> The competence required for the task is described in the corresponding expertise model, which serves as the basis for the decomposition of the sub-system design (implicit dependency).

Other sub-system implements Task (reference to Task Model):

> An Other sub-system is the actual realization of one or several tasks that have been assigned to an application. The Functionality slot of the Other sub-system will reflect the Description of the Task.
>
> The competence required for the task may be described in an expertise model. In that case it may serve as the basis for the decomposition of the sub-system design (implicit dependency). However, in general the Decomposition Paradigm of an Other sub-system can be anything.

Interaction sub-system implements Transaction (reference to Communication Model):

An interaction sub-system is the actual realization of one or several transactions in which the application is involved. The Functionality slot of the interaction sub-system will reflect the Description of the Interaction.

The transaction plan in which the transaction is involved is described in the communication model, which serves as the basis for the decomposition of the sub-system design (implicit dependency).

Platform has user Agent (reference to Agent Model):

The actual system has one or several users that are described in an Agent Model. It must be verified that the Agent model specifies the General Capabilities (in particular the vocabulary) that are required of a user. The Role slot of the Agent can be further specified to, for example, system maintainer.

Platform has system model Agent (reference to Agent Model):

The actual system is itself considered as an agent that is described in an Agent Model. It must be verified that the Agent model specifies the General Capabilities (in particular the vocabulary) that are supported by the application.

Hardware Configuration available in Computing Resources (reference to Organization Model):

The Hardware Configuration is described as a Computing Resource in the (target) Organization Model.

Software Configuration available in Computing Resources (reference to Organization Model):

The Software Configuration is described as a Computing Resource in the (target) Organization Model.

Environment available in Other Resources (reference to Organization Model):

The Environment in Platform Design refers to the physical location and circumstances of system use: office, desk, lighting conditions, accessibility, and so forth. These elements are described as Other Resources in the (target) Organization Model.

Chapter 3

States of the Design model

Walter Van de Velde, Cuno Duursma, Guus Schreiber

In this chapter we define and list useful states of the design model. The treatment of states so far in the project have been exploratory: the notion needs to be better understood by the work of identifying, using and validating states. We therefore start this chapter with a digression on the notion of state in general, and turn to design model states in the later sections.

3.1 The Idea of States

The development of a model is seen as pushing it through a sequence of model states, i.e., identifiable stages in the modeling process. Obviously one can think of many states: any partially developed model in principle determines a possible state. Many of these will not be useful, however. We therefore use the following notion of state:

> A model state is an identifiable stage in the modeling process that is an arguably useful step toward a stated objective.

There are two points of view one can take on the definition of usefulness, namely the perspective of model development and the perspective of the project management.

From a point of view of model development the states are those that one needs to go through in order to reach a filled in model. Here the states are mainly structural and ordered by a pre-requisite relation. This means that a state is a pre-requisite of another one if reaching the latter requires information that is available from the former. The states that can be derived in this way are a prototypical partial ordering on model development states. We call them the development-states. A development state is described in two parts: a reference to one or several model components, and one or several characteristics of that component. The "semantics" is that the state is reached for a particular model when the referred components satisfy the characteristics.

Model component *Obl.* a reference to an element in a model. There may be multiple components in one state, but at least one.

Component characteristic *Obl.* an identifiable characteristic of the component or components. There may be multiple such characteristics, but at least one.

A reference is by structural access relative to the top-level of the model using relations and at most one attribute at the end. This allows to refer to any component in a model in a strict way. A component is either defined in the model template or an element of a specific refinement of the model. For example a blackboard architecture design will refer to knowledge sources and blackboard organization.

The name of a component can be augmented with a qualifier that indicates its instances, in case there are more than one instance of the component in the model, or in case there are multiple instances of the model (e.g. initial or target).

Five basic component characteristics are being used:[1]

- *empty:* there is no information about the model component available
- *listed:* the possibilities have been listed, no choice is made yet but the options are clear.
- *identified:* the component has been identified but not yet described in detail
- *described:* the component is described in detail
- *validated:* there is strong evidence that the component description is adequate.

In order to reach a model development state one can use elements from the library, guided by modeling guidelines. A modeling guideline proposes part of a model as likely to be adequate for an application with a certain set of task features.

So far for the model development view. From the point of view of the project management the models need to be developed to the extend necessary and using the available resources. Given the objective of developing a model in a project the development states are a quasi unavoidable frame for work. The project management must modulate the sequence of states that are being reached based on information of the project situation, the product objectives and the risks.

From this point of view an objective is a quality objective for the product. This is in line with the CommonKADS quality viewpoint [Williams et al., 1992]. There elements of a quality model for a CommonKADS system are provided, based on an elaboration of ISO9126 quality standards [9001, 1987]. Such a quality model is a refinement hierarchy of which the top level qualities are the following:[2] functionality, reliability, usability, efficiency, maintainability and portability. An objective, in our view, is then stated in terms of these quality objectives. Of course, for an individual project the objective will be qualified in terms of more specific requirements and the different qualities may be prioritized.

Taking this quality viewpoint, the purpose of the CommonKADS methodology is to reach the stated objectives within the boundaries of available resources. In the modeling approach this means developing the models of the CommonKADS solution to a problem in

[1]These are the ones from the organization model, including the characteristic listed.

[2]The standard focuses on software. It may be that the CommonKADS product model requires an extension or modification of the software quality model. For the purpose of this document, since we are dealing with KBS design (mostly software), the model brings us a long way and we have not felt the need to change it in its most general (standardized) layers.

an organization such that, once delivered and installed, it satisfies the quality objectives. Obviously then, product quality must be related to model characteristics. More specifically it must be understood how a specific model characteristic contributes to a specific product quality. States, then, are useful because they establish model characteristics that contribute to the safeguarding of a quality objective endorsed by the project.

The states then require a more elaborate context, the risk management guideline. It includes, in addition to the target state:

Situation *Obl.* a description of an identifiable situation, referring to the information in the present product model.[3] The identification may require additional investigation.

Objective *Obl.* a quality objective that the state contributes to. There may be multiple such quality objectives, but at least one. (maybe this must be extended to a description of impacts).

Risk *Opt.* this is a more specific statement on how the situation, if not properly attended to, may lead to a product characteristic that has an impact on the quality objective.

It is important to understand the role and nature of these risk management guidelines. They are not telling what the contents of the model should be, but only what characteristics this contents should satisfy. The perspective of the project manager is not concerned with the details of the model. The model state is just a summary of the contents in terms of its usefulness to the overall project. Thus, the risk management guidelines are different from the modeling guidelines that are used in the libraries. In the libraries one finds guidance on when to use a certain (partial) model. For example one may find guidance on when to use this or that type of architecture or this or that type of representation. This is important from the viewpoint of the model developer who needs to bring the model in the states required by project management. The model developer has to fill in the model and may use the guidance he or she gets from the library. Note that this is not risk driven but task-feature driven. In Figure 3.1 the different elements involved in the modeling process are related.

In the following subsections we describe development states for the application, architecture and platform design. Later sections expand on the risk management guidelines.

3.1.1 Development States of Application Design

In this section we list major development states of the application design and provide a short rational for them. In Figure 3.2 the development states are related by a pre-requisite relation. This relation indicates that a state is a pre-requisite for another state in the sense that reaching the latter requires information which is available from the former. This also gives a rough ordering on states in time.

1. **Sub-Systems of Application Architecture Design: Identified**

[3] This is a deliberate restriction to scope the modeling guidelines. Other situations may be important but will to be detected on line as project management proceeds. For example the situation in which the expert becomes ill threats the correct development of the expertise model and therefore endangers the reliability of the product. This is realistic but out of our scope here.

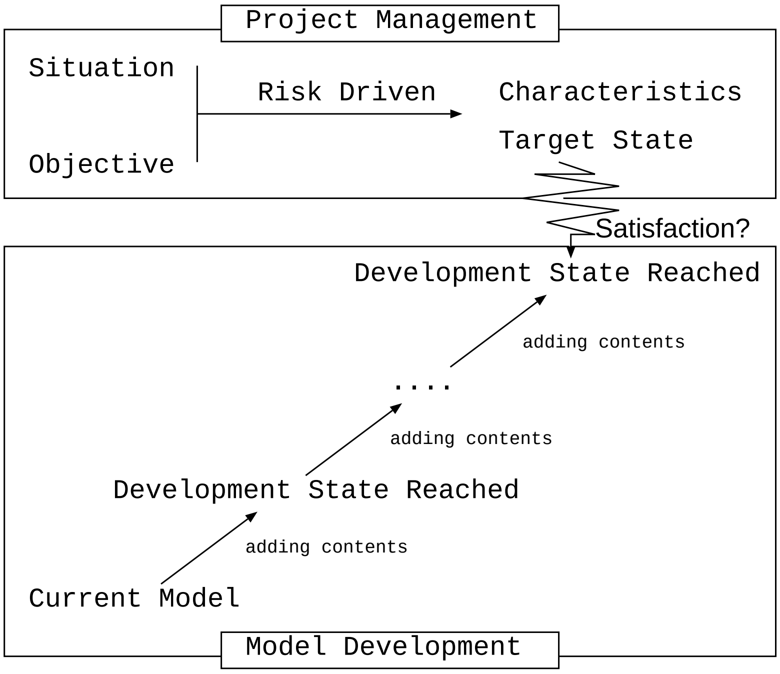


Figure 3.1: Risk driven project management sets up target states. These are satisfied by modeling activities, pushing the model through development states by elaborating the model. Risk management guidelines are important for project management. They set up target states. Modeling guidelines from the model library are used in model development. They propose contents for the models. The model development work itself is not risk driven, unless there is a nested cycle structure in project management. The purpose of the development work is to satisfy the target state set up by project management.

The sub-systems are identified and named. This gives a main decomposition of the design work and allows project management to focus on the most critical or difficult ones.

2. **Type of Sub-System: Identified**

   The type of a sub-system is a crucial indicator of how to approach its further design.

3. **Functionality of Sub-System: Described**

   The functionality of the sub-system needs to be consistent with the task model or communication model

4. **Decomposition Paradigm for Sub-System: Identified**

   The decomposition paradigm follows from the type. It is an important indicator for further design activities.

5. **Decomposition of Sub-System: Identified**

   This is the rough decomposition of the sub-system, without details and in an informal (loosely specified) language.

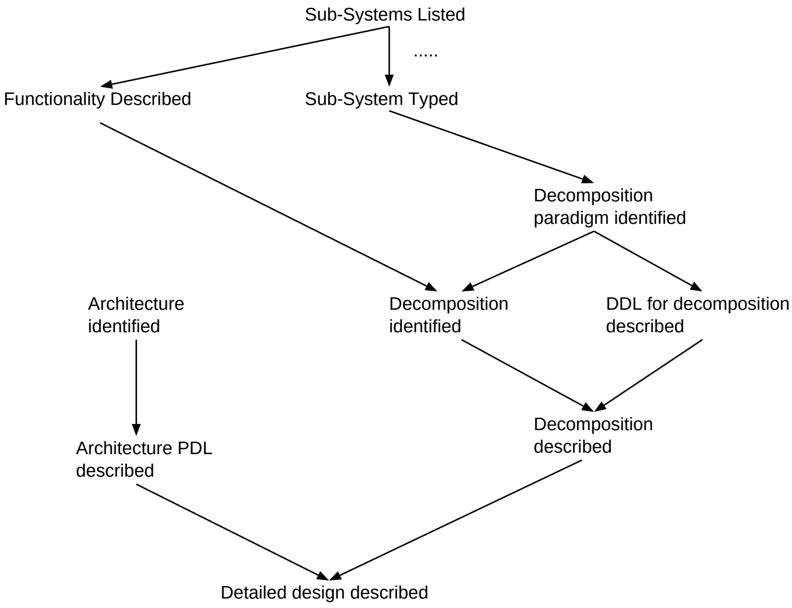


Figure 3.2: The development states for application design and the pre-requisite relation among them.

6. **DDL for Decomposition of Sub-System: Described**

   Given the paradigm and the idea of the decomposition a DDL is specified

7. **Decomposition of Sub-System: Described**

   The description of the decomposition within the selected DDL

8. **Architecture for Detailed Design of Sub-System: Identified**

   The architecture is identified based on the functionality and other constraints. This provides the first direction for the detailed application design.

9. **PDL for Detailed Design of Sub-System: Described**

   The PDL is needed for the detailed design of the sub-system within the identified architecture to start.

10. **Detailed Design of Sub-System: Described**

    The application design can be considered finished when the application architecture is completed and the detailed design of all sub-systems is finished.

3.1.2 Development States of Architecture Design

In this section we list major development states of the architecture design and provide a short rational for them. In Figure 3.3 the development states are related by a pre-requisite relation.

1. **Paradigm for Computational Architecture: Identified**

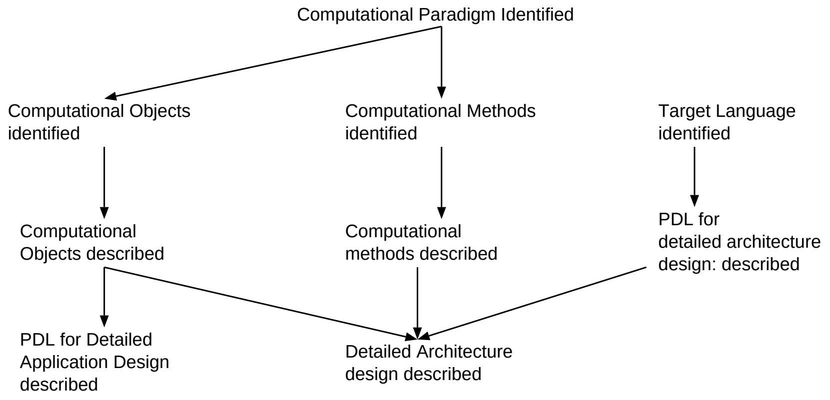


Figure 3.3: The development states for architecture design and the pre-requisite relation among them.

Typically a selection of a computational paradigm. This must be guided by the type of sub-system that the architecture is used for, its functionality and may play a role in the decomposition paradigm for the sub-system.

2. **Computational Objects: Identified**

   The identification of the computational objects gives a first indication of the kind of representations that will be supported by the architecture. This is important to judge (or assure) the match between an architecture and the sub-system it is intended for.

3. **Computational Methods: Identified**

   The identification of the computational methods is required for the detailed application design within the architecture to work. Without an understanding of the methods good design is hard to image.

4. **Computational Objects: Described**

   The kind of objects manipulated by the architecture are now detailed. This is a straight step toward the specification of a PDL for detailed application design

5. **PDL for Detailed Application Design: Described**

   The elaboration of a precise language for programming within the architecture, to be used by the detailed application design.

6. **Target Language: Identified**

   In case the architecture needs to be implemented then a target language is chosen as a step toward implementation decisions.

7. **Computational Methods: Described**

   Only required if the architecture needs to be implemented from scratch.

8. **PDL for Detailed Architecture Design: Described**

   The program description language for implementation of the architecture (often omitted in practice where straight implementation is followed)

9. **Detailed Architecture Design: described**

   The detailed architecture design is finished: a program that implements it is specified

3.1.3 Development States of Platform Design

In this section we list major development states of the platform design and provide a short rational for them. In Figure 3.4 the development states are related by a pre-requisite relation.

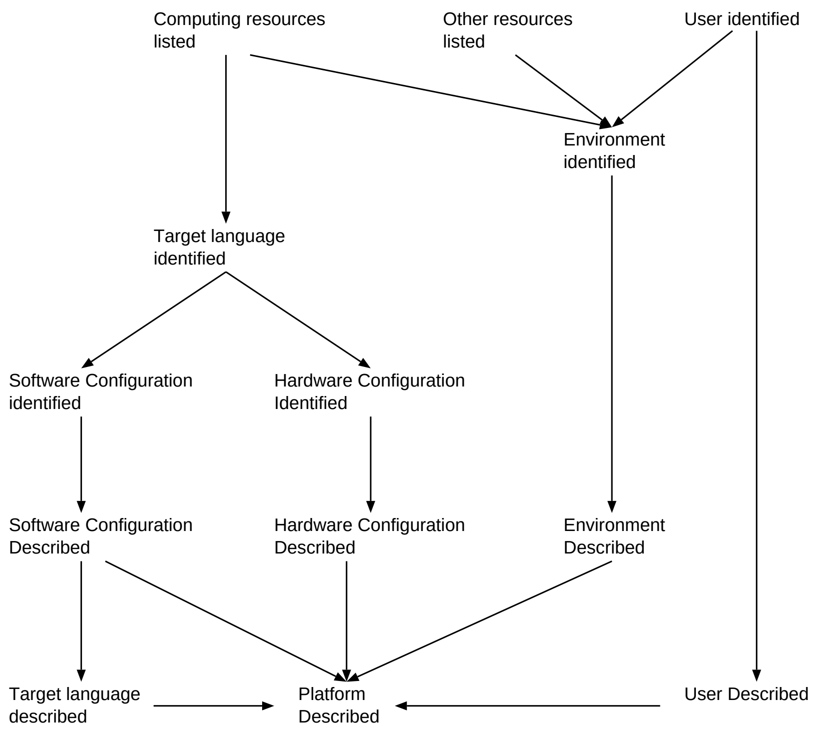


Figure 3.4: The development states for platform design and the pre-requisite relation among them.

1. **Computing resources listed** (state of (target) organizational model)

   an inventory of available computing resources to get a first idea on the kind of equipment and company policy.

2. **Other resources listed** (state of (target) organizational model)

   an inventory of other resources to get a first idea on the kind of user environment and company policy.

3. **User identified**

   the user is loosely characterized, giving an indication of the circumstances of use of the application

4. **target language identified**

   constraints from computing resources and other requirements from the architecture design provide the class of languages for implementation

5. **software configuration identified**

   available software or needed software is being identified based on available and requirements from target language(s)

6. **hardware configuration identified**

   available hardware or needed hardware is being identified based on available and requirements from target language(s)

7. **environment identified**

   hardware configuration, other resources and user identification give a first complete picture of the circumstances of use of the application

8. **software configuration described**

   elaboration of identified software configuration as a basis for installation or purchase

9. **hardware configuration described**

   elaboration of identified hardware configuration as a basis for installation or purchase

10. **environment described**

    elaboration of identified environment as a basis for installation or purchase

11. **target language described**

    the detailed description of target language so that a PDL can be derived for purposes of detailed architecture design

12. **user described** (elaborate state of Agent Model)

    detailed description of user capabilities in line with other design decisions

13. **platform described**

    the platform design can be considered finished if all the pre-requisite states have been reached

3.2 Risk Management Guidelines

In this section we provide a list of risk management guidelines that involve design. The guideline is to be used in the first place by the project manager to set short and medium term project objectives. They characterize situations that may affect the necessity or desirability for the design model to be developed in a certain state.[4]

In order to describe risk management guidelines the following format will be used:

[4] The origin of this section is in a questionnaire that was put together by Clive Bright (LR) to initiate the investigation of concrete states, risks and quality characteristics.

Situation *(Obl.)*: a paradigmatic situation that a project may find itself into. The situation refers to the project in the widest possible sense and includes characterization of the application domain and task itself as well as of the project (e.g., resources).

Objective *(Obl.)* one or more objectives for the final product (the CommonKADS solution). These are stated preferably in quality terms and may be more or less specific. Remember that quality is a product feature with respect to the end users.

Risk *(Opt.)* a description of a characteristic of the project that is plausible to arise in the situation if not properly handled and that constitute a threat on reaching the objective.

Target state *(Obl.)* a state of the product model that, when reached, eliminates or at least reduces the risk and therefore has an arguable effect on the project objectives.

Rational *(Opt.)* an explanation of why reaching the state eliminates or reduces the risk of threat on the objective in the given situation. Quotes and references can be useful here.

Of these elements the situation, objective and target state are obligatory. The other elements are often obvious, and therefore left implicit, but sometimes they are not. The risk and rational are also important for the project manager to assess whether a guideline is applicable or not.

Below is a preliminary list of modeling guidelines described in the format above.

1. **Situation** Multiple programmers involved

   Objective high quality software (i.e., bug free and maintainable)

   Risk inconsistent data model and redundant functionality

   Target state application design functionality described

   Rational if the interfaces are rightly specified then conflicts between pieces of code are less likely. Such conflicts tend to create bugs and decrease the maintainability of systems. The same holds for redundant functionality.

2. **Situation** target language is a low-level general purpose language.

   Objective maintainable system, explanation facility

   Risk the resulting system will have a complex design and implementation, making the system un-maintainable.

   Target state computational architecture (preferably of abstract level) described and its PDL, used as the formalism for programming the application, is described

   Rational a high-level computational architecture helps one to focus on the epistemological issues when coding, rather than the computational ones

3. **Situation** the task that is to be supported is prone to change over time.

   Objective The delivered product must be easy to maintain and be adaptable to expected new requirements

Risk The product will not be maintainable, due to inflexible design

Target state application architecture described

Rational if the application is built on an architecture that provides good generality and flexibility and is designed for this *type* of application, then the application can be maintained without changes to the architecture support

4. **Situation** implementation of artefact will be done by third party or client himself, based on design delivered by this project

   Objective the actual implementation work should proceed in an orderly way, and implementation risks should be minimal

   Risk contractual quality of software specification not met

   Target state the complete design model shall be developed, and documented

   Rational a prerequisite for a smooth implementation job is that the design is regular and is well documented with respect to rationale for made design decisions

5. **Situation** the application to be developed will be part of a group of cooperating systems supporting an organisation

   Objective the application should be easy to plug into an environment with other systems

   Risk integration with existing other software systems will fail, and/or upgraded versions of other software systems will fail to be integrated

   Target state design model must be developed, especially in the interface to the other software

   Rational A well-designed and implemented interface component critically determines whether integration will cause problems

6. **Situation** the client intends to use the system in a context where there are strict requirements on response time

   Objective the design should guarantee that performance requirements will be met

   Risk required performance characteristics will not be met

   Target state in the architecture design, the critical computational methods must be described. These methods are the ones that either process large volumes of data or performs complex processing

   Rational volumes of data and complex processing are the typical reasons for bad performance

7. **Situation** functionality is not well-specified, they seem to be too broad, and maybe inconsistent

   Objective understand what the known requirements imply with respect to the functionality of the application

   Risk it is impossible to design an application for the requirements

   Target state functionality of sub-systems in the application design is described

Rational performing application design is equivalent to analysing the requirements and breaking it down into understandable and verifiable sub-requirements

8. **Situation** application is to be used in a multi-user mode, with several concurrently active users

   Objective the application must be reliable and secure

   Risk multiple concurrent users may cause the application to crash, or produce execution-context dependent output, or compromise security standards

   Target state in platform design, software configuration is described, and in architecture design, the paradigm for computational architecture must be identified. This is especiually critical for the area of data management

   Rational the architecture should isolate the application from all knowledge about concurrent users. The data management aspect is typically where concurrent users may be able to interfere with one another

9. **Situation** application will run on a platform with sophisticated user-interaction technology

   Objective the application should be easy to use, and predictable

   Risk the user will be confused during interaction, due to the application not exhibiting a predictable behavior, and the interface technology is used by the application in unexpected ways

   Target state the interaction part of application and architecture must be described, and the software/hardware configuration of the platform must be described. It is important to verify that the interaction will behave according to the standards set for the organisation

   Rational the comunication part of the design model is detailed enough to support validation of the behavior of the application.

10. **Situation** the application will be used in highly critical contexts

    Objective the application needs to be correct and reliable

    Risk the application might turn out to be unreliable

    Target state all parts of the design model must be described in full detail

    Rational to be able to verify the design w.r.t. application correctness, a strict and detailed description is needed.

11. **Situation** new platforms are expected to be used for the application

    Objective to have the application available on new platforms, quickly and at low cost

    Risk the porting of the application might be very costly

    Target state application part of the design model should be elaborated to the state PDL described. The architecture part of the design model should be developed to the states computational objects described and computational methods described. The platform part of the design model should have target language identified, and software configuration identified.

Rational porting is easier if the system was built on a small kernel, which the implementation of the architecture. Porting this kernel will make the application run on the new platform

12. **Situation** available requirements are ambiguous

    Objective availability of an application that satisfies the users' needs

    Risk the application will not match the expectations of the users

    Target state do not start or continue design work until requirements have been clarified

    Rational design is a resource-demanding process. It is unwise to spend resources on building an application that eventually turns out not to be the correct one. Requirements may, during the design process, be detected not to be as clear as was believed.

13. **Objective** verify that an application can be designed in practice

    Risk the resulting design will not work as required, or will be expensive to maintain, or will never be finished/delivered

    Target state architecture described

    Rational a divide and conquer approach to design, where the intermediate architecture provides visibility to the problems involved, and increases the knowledge on design for this type of application

14. **Situation** available languages for implementation are not well adapted to the needs of the project

    Objective to have an application up and running quickly at low cost

    Risk the mapping of the functional descriptions to the languae will be complex. This can cause the application to be buggy, and will have a negative impact on maintainability

    Target state in the architecture part of the design model, describe the PDL for detailed application design.

    Rational Designing a new lanuage that matches the requirements of the application will simplify the application design. This new language will then be mapped to the languages available on the platform.

15. **Situation** the constraints frequently change in the environment where the application is to be used

    Objective the application should always be useful, in the sense that it should behave in accordance with the latest norms, standards, and conventions

    Risk the application will be costly to maintain

    Target state in the architecture part of the design model, identify the computational objects and methods.

    Rational designing an architecture that is general enough to be the host for many applications of a certain type, will simplify adaption of the actual application. What we aim for is an architecture that not only supports the application as we now know it, but will support a *family* of applications.

16. **Situation** the environment where the application is to be used will frequently undergo changes. New demands on the application behavior, changes to the platform, new software systems to cooperate with.

    Objective application needs to be consistent with the current environment

    Risk application will be buggy due to constant maintenance, the design and implementation will become unmanageable.

    Target state the design model must be described in great detail, in application, architecture and platform parts.

    Rational explicit separating of application from architectures from languages makes it possible to perform separate verification and validation on these entities

17. **Situation** task-specific shell available for use in the project

    Objective application should be quickly available for use

    Risk resource or time over-run in design and implementation work

    Target state do not develop architecture and platform parts of the design model. Only develop the application part to the states type of sub-system identified and functionality of sub-system described.

    Rational if a suitable task-spoecific shell is available, then structured knowledge can more or less directly be put into the shell, and hence an application will be available. There is still a need for a 'design description' of the way that the shell is used for the current application.

18. **Situation** the project is planned to small and short-lived

    Objective perform feasibility demonstration of application

    Risk cost and time over-run in building the demonstrator

    Target state develop the design model only to the states of subsystems of application architecture design identified, computational objects and methods in architecture design identified, and in platform design, target language identified.

    Rational if the project has been allotted only very little resources, then it is not sensible to embark on a full-blown design process.

19. **Situation** the application seems to be easy to implement directly in some standard language

    Objective the application should be reliable and maintainable

    Risk maintenance will be difficult to perform in a reliable way

    Target state the application part of the design model should be pushed into the states decomposition paradigm identified and functionality of sub-systems described.

    Rational by decomposing the application it is possible to make more informed judgements about the complexity of the design and implementation

A major problem with larger collections of guidelines is to find the relevant ones. Design guidelines are indexed by two elements: the situation and the objective. For objective we can use the quality framework kind of descriptions. For the situation there are two types. Project situations refer to aspects of the project, in the sense of personnel involved, timing constraints, resources and so on. Task features are another way to characterize situations. They are related more to the characteristics of the domain of application. When the set of guidelines grows too large then a more precise categorization of situations may prove necessary.

Chapter 4

Design model notations

Peter Terpstra

In this chapter we define a specific notation for documenting the major design decisions in the CommonKADS design model. The main areas of interest are the application specification for KBS and interaction sub-systems, and the architecture specification.

The notation (HADL, High-level Application Design Language)is specific for CommonKADS, as it is heavily based on the structure of the CommonKADS model of expertise. The approach taken is that the design notation should clearly reflect the design decisions involved in operationalising a model of expertise.

This offers a desirable separation of concerns: separating on the one hand the mapping from conceptual problem-solving descriptions to design structures, and on the other hand refining a design. The first part, the mapping, is the difficult and critical step, and needs to be documented in a explicit and unambiguous way. As the second part is mainly concerned with elaborating on a given design, it is more manageable.

This is not to say that design refinement and elaboration is a trivial task. We should not underestimate the criticality of it. But in the context of the CommonKADS methodology, the first part is the most important, and as it is unique for the methodology, it merits some attention. Design refinement can be managed and performed in different ways, and is less dependent in the methodology.

For these reasons we will provide a notation suitable for high-level design. The notation is presented in the rest of this chapter. The chapter is extracted from the document "CommonKADS specific design decisions and their notation" by Peter Terpstra (KADS-II/M7/UvA/043/1.1). That document also includes two large examples, illustrating the design decisions involved, and how they can be represented in the design notation.

4.1 The concept of high-level application design

Given a knowledge level model on the one hand and a symbol level architecture on the other hand, we have to define the relation between them. In the design model, this is called high level application design.

The bulk of this chapter is on the application design of the inferences and the relation between the inference layer and domain layer. In an expertise model the algorithms which are used for implementing the inference are not specified. Moreover, to enhance reusability, the inferences in the expertise model do not make a commitment to the representation of the data on which they operate. Therefore they refer to the domain knowledge in an indirect way via roles and model schema's. The CML leaves the specification of the relation between the domain layer and inference deliberately underspecified. $(\mathrm{ML})^2$ defines this relation as an object/meta relation via a rewrite grammar. This method of instantiating data and knowledge as input and output for programs is not often used in traditional software specification or programming languages[1]. The indirect mapping between inference layer and domain layer via roles introduces a lot of design decisions.

The operationalisation of the task layer on the other hand, can, as we shall see, be relatively straight forward. We will not elaborate on the design of domain layer constructs. Not because this is such a trivial task, but because the optimal solutions are very dependent on the chosen architecture and as such not the subject of the high level application design.

Section 4.2 describes the notation as used for the design of the task layer. Section 4.3.1 describes the notation for the operationalisation of inference roles. Section 4.3.2 gives some examples of how role mappings can be operationalised and criteria to choose between them. Section 4.3.3 enumerates a set of design alternatives related to the operationalisation of the inference steps inside an inference.

4.2 Task layer

In the *Common*KADS expertise models we can make a distinction between global control and local control. Global control is the control in the task layer. Local control is control inside inferences. Because *Common*KADS structures the knowledge base into three layers, in general the global control is not very complicated. One only has to specify how often and under what conditions tasks or inferences are invoked. More specific detail is hidden in the control of the inference methods. Therefore one of the characteristics of a *Common*KADS expertise model, is that the global control loop is described explicitly. Moreover the task layer algorithms in *Common*KADS systems are often well defined, especially in problem solving methods. This type of specification is most easily described in an imperative language. Therefore the task layer in the CML is an imperative one. This is in contrast with other approaches where control is specified by means of relative preference criteria (agenda based systems), trigger conditions (blackboard systems or finite state machine), goals (state space search) etc.

In structure preserving design one tries to mimic the structure of the expertise model during the operationalisation. For the design of the task layer this means that one keeps the global control explicit in the task layer and does not distribute this to the inferences themselves.

The expertise model only gives an idealized specification of control. As is shown in [Terpstra, 1993] one often has to introduce auxiliary tasks for initialisation or bookkeep-

[1]The existence of an explicit inference layer is also not found in other knowledge based system development methodologies (cf. Components of Expertise [Steels, 1990], Role Limiting Methods [McDermott, 1989]). These systems do not make a distinction between tasks and inferences, but tasks are invoked directly on domain knowledge.

ing. Also it may be convenient to introduce auxiliary parameters which are used to pass information when this information otherwise would have to be recomputed in the different inferences or each time a particular inference is invoked.

Because the *Common*KADS CML and $(\mathrm{ML})^2$ specifications of the task layer already have a procedural type of notation, we have adopted these notations, but have added facilities to introduce type information for roles and a refinement of slot, which specifies which expertise model task is refined.

This gives us the following structure for a HADL task. We do not prescribe the exact syntax of the expressions in these task specifications. Given the type of input for the design this could be similar to the CML or FML or whatever is appropriate:

Task Module

- **Name**
- **Refinement of** The expertise model task, which is operationalised in this task.
- **Task Definition**
  - **Goal**
  - **Input** A list of input roles (cf. input parameters).
  - **Output** A list of output roles (cf. output parameters or function results).
  - **Spec** A textual description of the task.
- **Task Body**
  - **Type** composite or primitive. A primitive task activates only inferences. A composite task may invoke other tasks.
  - **Subtasks**
  - **Additional Roles** A list of roles, which are only visible in the task (cf. local variables).
  - **Control Structure** The body of the task, described as a pseudo algorithm in an imperative language.

End Task Module

4.3 Inference layer

In the expertise model an inference consists of a name, operation type, input roles, output roles, static roles and the inference specification. The operationalisation of an inference in the expertise model is specified in an inference procedure which contains the operationalisations of the role mappings and the inference specification. Using the design guideline to make the link between the expertise model and the design model as explicit as possible, we define for each CML/FML inference slot a corresponding slot in the inference procedure. So an inference procedure contains entries for a name, input roles, output roles, static roles. The operationalisation of the inference itself is denoted by an inference method, the algorithm or operation which realises the inference specification.

For each of the *Common*KADS specific constructs we describe some alternatives for operationalising it and the criteria to choose between them. Each set of alternatives introduces a design decision and a corresponding notation is added to the HADL. Therefore the sections on the inference layer not only provide a specification of the HADL, but also a set of guidelines to choose between different design alternatives.

4.3.1 Inference Roles and role mappings

Not all roles are inference specific names for domain knowledge entities. There are roles which are linked to the domain layer through a role mapping and roles which are the result of an inference and for which no direct mapping exists. Figure 4.1 (adapted from [van Harmelen *et al.*, 1991]) shows the difference:

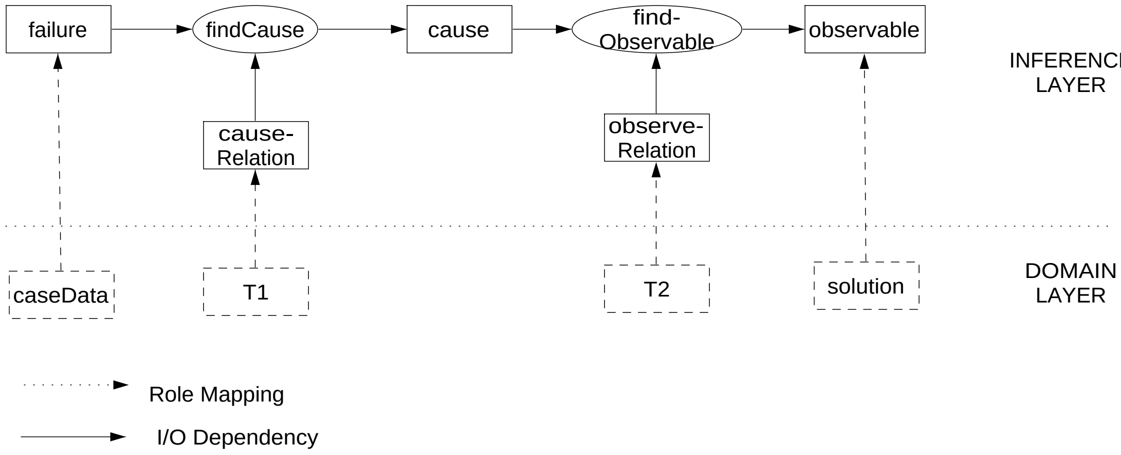


Figure 4.1: Knowledge roles as mapping or I/O dependencies

The roles failure, cause-Relation, observeRelation and observable are linked to the domain layer through a mapping. Static roles such as causeRelation always have a mapping. The role cause, which is an output role of findCause and input for findObservable is not directly linked to the domain layer, but is an I/O dependency between inferences[2]. Such a dependency is typically operationalised by input and output parameters of inferences, which are linked together on the task layer.

This gives us the following distinctions wrt. the input and output of inference procedures: Each role is denoted by a variable. We use input and output parameters to denote the roles which are not connected via a direct mapping to the domain layer. Roles which are linked to the domain layer are denoted by domain input variables or domain output variables. An inference procedure may also contain local variables, which are not used for communication between inferences or for roles.

[2]The instantiations of such a role as a whole do not have a direct mapping to the domain layer, but parts of those instantiations could be linked. Eg. in the example of figure 4.1 a cause might be (cause 'battery-low 'car-does-not-start). The arguments of this term have a link to the domain layer via the roles cause-relations and failure. The term as a whole does not have a meaning on the domain layer.

parameter-definitions	::=	input-parameters from-domain-input-links output-parameters to-domain-output-links local-variables	
input-parameters	::=	**parameter-input:**	⟨parameter-declaration⟩⋆
from-domain-input-links	::=	**input from domain layer**	⟨parameter-declaration link-definition⟩⋆
output-parameters	::=	**parameter-output**	⟨parameter-declaration⟩⋆
local-variables	::=	**local variables**	⟨parameter-declaration⟩⋆
to-domain-output-links	::=	**output to domain layer**	⟨parameter-declaration link-definition⟩⋆
parameter-declaration	::=	*variable-name*	**type** *parameter sort*

For the specifications of domain/inference layer links we first have to discuss different operationalisations of role mappings and how these mappings can be used in an inference procedure (section 4.3.2). Their BNF definitions can be found in section 4.3.2.3.

4.3.2 Designing Role Mappings

In *Common*KADS expertise models the relation between inference layer and domain layer is indirect via a role mapping. In general these mappings are defined between domain and inference sorts[3]. This indirect communication between layers introduces design questions which are not often found in more traditional software engineering.

Some characteristics of and requirements on the implementation of a role mapping are

- The unique reference constraint.

  A general constraint on the expressiveness of the mapping is that each inference layer name (role instantiation) points to an unique domain layer construct cf. [van Harmelen, 1992, van Harmelen *et al.*, 1992]. In other words the mapping should be a function from inference layer constructs to domain layer constructs (ie. an injective relation between domain layer and inference layer).

- Whether syntactical transformations are applied during the mapping.

  Some mappings not only map domain layer constructs as a whole, but manipulate the element of the constructs in a certain way. Eg. $\mathsf{A} \rightarrow \mathsf{B}$ maps to $\mathsf{observation}$ (B ,A). A and B are not only renamed at the inference layer, but also their order is changed.

[3] although there are examples where these mappings are direct relations between domain layer and inference constants.

A facility for changing the structure of domain layer enhances reusability of domain knowledge in the sense that the knowledge can be transformed into an inference specific representation. As we shall see below not all realisations of the mapping can deal with this type of manipulation.

- One has to be careful that the set of terms of the inference layer is kept disjoint of the set of domain layer constructs. Otherwise this could introduces several complications such as the possibility that entities are confused with their names or that inference layer relations (sets of sets of tuples) are unified with object layer relations (sets of tuples). There are several solutions to realise this eg.:
  - All domain layer constant symbols are quoted. All domain variables get a gensymed inference layer name. This is the $(\mathrm{ML})^2$ solution.
  - Dividing the set of inference layer terms into "pure" inference layer types and types which can be instantiated with names of object layer constructs. The inference layer interpreter prevents unification between these types.
  - Using a syntactical constructs: for example domain layer names occur on special places in the inference layer predicates arguments.
- The communication between the two layers involves both a transformation, which transforms terms from the language of one layer to another and a semantical mapping, which denotes the relation between truth values of the original and the transformed expressions.

  A design decision is whether one implements these syntactical and semantical aspects of the mapping in one mechanism or whether they are kept distinct. Keeping them distinct makes it easier to use one syntax mapping definition for different semantical mappings.

4.3.2.1 Some alternative mapping formalisms

Below are some candidates for the realisation of the mapping relations.

- Parametrisation of definitions of primitive inferences by domain layer sorts.

  The formal parameters of the inference procedure are instantiated during compile time by domain layer sorts. An example of this type of mapping can be found in KbsSF ([Jonker & Spee, 1992]).

  Because one formal sort can be instantiated during reasoning only by one domain layer sort at a time, these mappings guarantee the requirement that an inference layer name points to only one domain layer construct.

  In KbsSF the relation between formal and actual sorts is one to one. Manipulation of the entities inside the parameters is not directly possible in the mapping function.

- Subsumption of concepts which are specific for problem solving methods.

  An example is the work on Problem Solving Method's (PSM's) in [vanWelkenhuysen & Rademakers, 1990].

PSM's which communicate with the domain layer, do this by referring to generic domain concepts. During modelling application dependent domain concepts are defined as subconcepts of the generic concepts used in the PSM's.

Eg. a method needs domain concepts which fit the generic domain concept **functional component**. In the TroTel [vanWelkenhuysen & Rademakers, 1990] application the functional component is further specialised into the concept **Rom**.

So the mapping is similar to the aforementioned parametrisation of inferences. The unique name requirement is guaranteed in a similar way. Manipulation of components of concepts is not directly possible. When the same domain knowledge is needed for different inferences, the knowledge could have to be represented in different formats. Or the PSM would have to be redesigned in such a way that it fits the application knowledge.

- Indexing of domain layer sorts by inference layer sorts.

  Typically this is implemented by domain-inference indexes which link inference layer sorts and domain layer sorts. An example of the usage of this indexing can be found in the operationalisation of the Sisyphus application [Schreiber, 1992] or [Duursma *et al.*, 1992] chapter 5[4]. The keys in the index tables are inference layer names, which point to domain layer constructs.

  It is easy to check whether the unique reference constraint hold. Manipulation of the entities inside the lifted domain layer constructs is possible if one writes the index tables in such a way that parts of the keys contain variables, which are instantiated by the matching mechanism.

- Rewrite grammars as used in $(ML)^2$.
  During reasoning the grammar is parsed and either inference layer constructs or domain layer constructs are instantiated.

  For this class of grammars it is decidable whether the references to domain layer are unique. The grammar mechanism is very flexible because all language elements are individually accessible and can be manipulated during the parsing.

  An advantage of rewrite grammars is that they can be written in a more compact way than lookup tables by using grammar rules which refer to other grammar rules. Being more compact, they are easier to maintain. But this recursive parsing of the grammar may introduce some inefficiency.

  Some of this inefficiency could be reduced by compiling them into index tables (cf. chapter 6 in [Duursma *et al.*, 1992]). So a possible design route is to specify the mappings in a rewrite grammar and compile them in a lookup table in a next design step.

- Mapping by deduction.

  In KARL [Angele *et al.*, 1991] Horn clauses are used to implement the mapping. This is possible because KARL does not make a logical distinction between inference and domain layer in contrast with for example $(ML)^2$.

[4] Actually the procedures used by Schreiber are more expressive than just indexing.. Some implicit domain layer reasoning is used on transitive and communicative relations.

Using a less restrictive formalism, the user has to be more careful than in the other approaches that the unique reference constraint holds. Moreover, deduction can be less efficient than the other mapping formalisms.

- In Desire [van Langevelde *et al.*, 1993] a transformation between reasoning modules is defined in transformation modules. Tables which link objects, sorts and terms define the transformation of expressions from one language to another (syntactical transformation). The constructs for these mappings are similar to $(ML)^2$ lift rules, but without a facility for recursion. Atomic sentences are mapped via a transformation which is both syntactical and semantical ie. the transformation also defines the truth value of a transformed atomic sentence.

4.3.2.2 Naming of entries in non declarative languages

The above mappings all assume that the domain language is declarative and that the semantics of a domain layer sentence is locally defined ie. can be constructed out the syntax of the sentence, without involving any computations or making references to other parts of the knowledge base, which are not explicitly mentioned in the sentence. This makes it possible to get access to the contents of a knowledge base, by only using the syntax as defined in a signature or ontology.

A lot of representation languages do not have this characteristic. For example frames in frame languages often contain slots which refer to procedural knowledge (eg. `if-needed` or `if-removed` slots). Multiple inheritance is another example. The syntax of a frame itself does not give clues about all possible slots in a frame. One has to take into consideration the definition of the "inference rule" for multiple inheritance.

This problem can be avoided when we define an intermediate query language, which gives a declarative abstracted view of the underlying language [Terpstra *et al.*, 1992]. The role mapping is then defined on this intermediate language and the underlying computation is hided by the query language interpreter. For example the syntax of a frame in the intermediate language would contain all its inherited slots.

4.3.2.3 Specification of mapping in inference procedure

In an inference procedure we not only have to specify the mapping formalism, but also how a mapping is used to instantiate a domain role variable. One has to specify

- the domain layer module from where the mappings get the domain constructs,
- the exhaustiveness of the invocation of the mapping function ie. whether it returns all possible bindings or only the next new binding,
- the direction of the mapping. Is it used to move information from object to inference layer or the other way around.

Now we are able to summarize how a domain role variable is instantiated by the role mapping.

link definition	::=	**role type**	⟨ dynamic | static ⟩
		from	*domain partition name*
		mapping	*mapping name*
		formalism	*formalism name*
		direction	⟨ **up** | **down** | **both**⟩
		exhaustiveness	⟨**one** | **all**⟩

4.3.3 Locus of action of inference methods

An inference method is a procedure which implements an CML or $(ML)^2$ inference specification. In general such a procedure consists of an algorithm working on a data structure (or an interpreter working on a knowledge representation).

*Common*KADS knows two different views on when an inference is primitive and should not be further decomposed.

1. The canonical view enumerates a set of primitive inferences from which all primitive inferences should be constructed. For example in KADS-I this set was defined in terms of sensible manipulations of the modelling constructs of the domain layer CML [Breuker *et al.*, 1987].

2. A more pragmatic view on primitive inferences defines an inference to be primitive when it is not interesting for the application to look at the internals of the inference. For example $(ML)^2$ knows the construct ask-prove, which is used to call a domain layer interpreter. Of course the workings of the interpreter could be simulated on the inference layer, but often this would result in an overly detailed inference specification.

Both approaches don't exclude the possibility that although an inference is called "primitive" that the computation behind the inference is complicated. For example the KADS-I canonical primitive inference match compares two structures and returns their differences. Comparison of two structures is equivalent to graph comparison, a problem which is in principle NP-hard.

The CML hides this domain layer computation. When we take the definitions in the CML literally then all inferencing is done on the inference layer [5].

In the following sections we will explore some alternative interpretations of inferences as denoted in the CML. We will make a distinction between domain layer reasoning and inference layer reasoning. In inference layer reasoning the input and output of the inference methods is inference layer data or knowledge, ie. domain knowledge/data which is selected from the domain layer and subsequently renamed into inference layer terminology via a mapping. In domain layer reasoning the inference method directly works on the domain layer knowledge/data. This distinction gives the following possibilities.

1. In pure inference layer reasoning the inference method gets all its knowledge/data from the inference layer.

[5] This is in contrast $(ML)^2$ which makes an explicit distinction between pure inference layer reasoning and inferences which construct a domain layer goal, invoke the domain layer interpreter and lift the answer of the domain layer upwards (ask-prove).

2. Mixed domain/inference inference layer uses some input from the inference layer, which is mapped to the domain layer and the inference methods work directly on the domain layer representations.

3. Pure domain layer reasoning works directly on domain knowledge/data.

4.3.3.1 Pure inference layer reasoning

In the KADS-architecture (figure 4.2) as proposed in [Duursma *et al.*, 1992] one of the design goals was to keep the distance between the expertise model and the design as small as possible. A conclusion which was drawn from this requirement is that almost all inferencing is done at the inference layer. The domain layer was represented in a domain representation language which was similar to the CML, but had a more formal syntax.

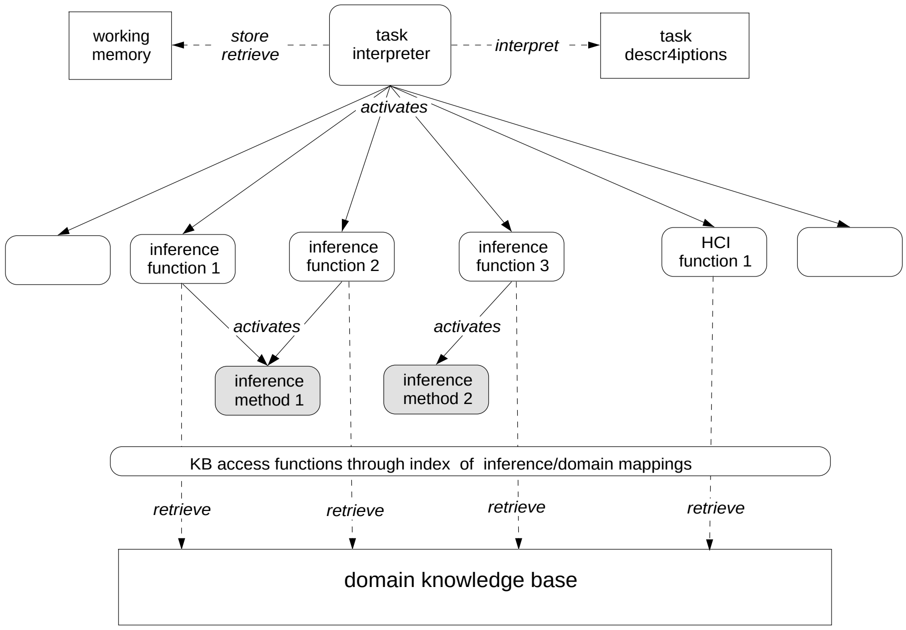


Figure 4.2: KADS architecture for pure inference layer reasoning

Such an architecture implies the following mappings and transformations (figure 4.3).

1. During design the CML is (trivially) mapped on the domain description language.

2. During reasoning two mappings could be involved:

   (a) Domain constructs are rewritten to their inference layer representations

   (b) A lot of methods use their own representation, which is optimised for the algorithm of the method. So in these cases the generic inference layer representation has to be mapped on the method specific data structures.

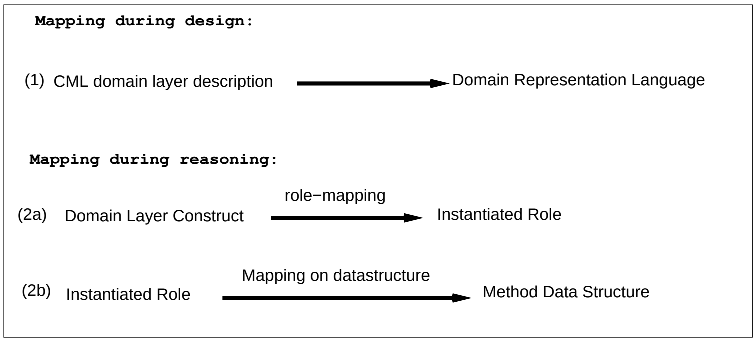


Figure 4.3: Mappings involved in pure inference layer reasoning

4.3.3.2 Mixed domain layer and inference layer reasoning

An alternative is to make inferences on the domain layer. Communication between inferences is on the task layer. This solution involves more mappings during design, but less during reasoning (figure 4.4). During design one has to define how the CML domain layer is mapped on the method representation. Secondly the role mappings have to be redesigned so that they directly link the method representation and the inference layer. During reasoning the domain layer method representation is linked to the inference layer via the method-specific role mapping.

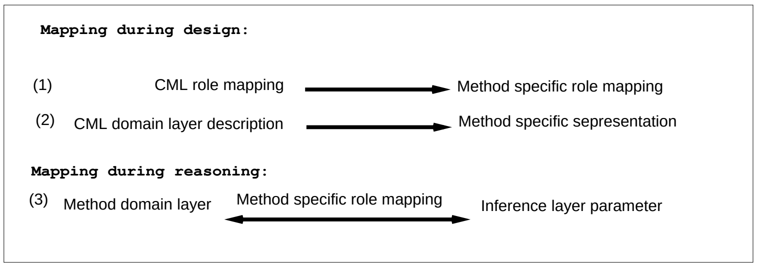


Figure 4.4: Example of mappings involved when inference communication on the inference layer and method works on the domain layer

4.3.3.3 Pure domain layer inferencing

In the former approach the inferences were made at the domain layer but they communicate at the inference layer by means of input and output parameters. In some applications in which a lot of data are involved this still may be an unattractive solution.
For example a lot of problem solving methods have a `generate and test` pattern. First all candidate solutions are generated, which are then filtered by a test, using some background knowledge. This could involve renaming the contents of whole databases at the inference layer cf. the Fraudwatch application [Killin, 1993].

Pure domain layer reasoning involves the following design steps and mappings:

- The inference structure serves as a template for the structure of the knowledge base and the inference methods.

- During design the CML-domain layer specification has to be mapped on existing data and knowledge bases.
- Wrt. the design of the inference method we can make a distinction between two cases: (1) The algorithm works on a general purpose domain knowledge representation eg. obtaining the value of an attribute in a frame language. (2) The algorithm works on a special purpose data structure, eg. a sort algorithm which uses as input pairs (key item) or matrix multiplication algorithm, using sparse matrices.

  In the first case design involves a formulation of inference layer inferences in terms of the predefined domain layer inferences. In the second case the inference methods have to designed from scratch.
- During reasoning there is no inference layer specific mapping involved (but explanation facilities are more complex, section 4.3.3.5).

4.3.3.4 Refined KADS architecture

The different types of invocation of an inference could lead to an architecture as in figure 4.5, where inference procedure1 works on the inference layer, inference procedure2 works on the domain layer (inference method2) but the output of the inference is still accessible on the inference layer and inference procedure3 works on the domain layer.

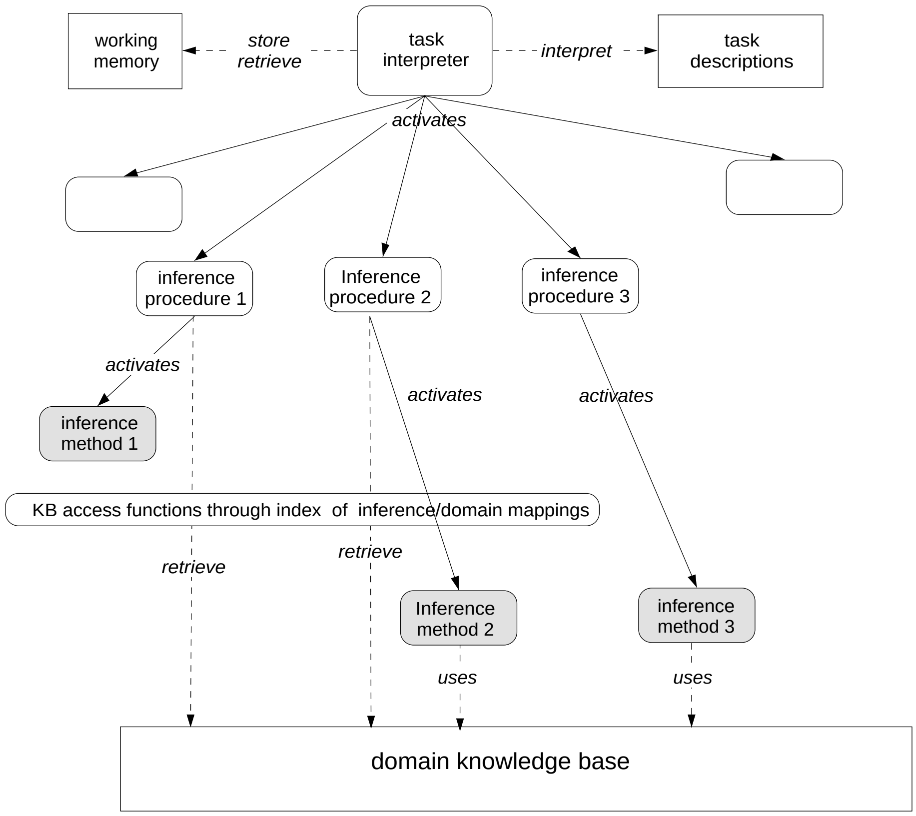


Figure 4.5: Architecture with methods working on different layers

4.3.3.5 Comparing the options

The criteria to choose between these options are

- reusability of domain knowledge:

  One of the goals of *Common*KADS is to keep the domain layer representation as reusable as possible. Therefore a domain model should be formulated independent of its usage. A conclusion which is often drawn from this (as in figure 4.2), is that inferencing should be done as much as possible on the inference layer. Then role mappings make it possible to use the same domain knowledge for several different inferences.

  This choice is valid, if only knowledge or data is used, which is collected following the *Common*KADS methodology. This is not the case for example, when existing databases are used as domain knowledge. In these cases it may be easier to re-formulate the inferences in terms of the existing representation (pure domain layer reasoning) or adapt the role mappings in such a way that they transform incoming data into a form suitable for the methods defined on the existing knowledge representation.

- reusability of inference methods:

  Pure inference layer and mixed inference/domain layer reasoning enhances reusability of inference methods, because the knowledge and data are mapped into or are already in a representation which fits the inference method.

  Pure domain layer reasoning may involve re-implementation of inference methods. As argued above, high level knowledge representation environments already have a set of predefined inferences. If the inference specifications of the CML map on these predefined inferences then reusability of inference methods in maintained.

  In pure domain layer reasoning the distance between the knowledge level model and the design model is larger than when reasoning is done at the inference layer. But because the knowledge level description still can be reflected in the structure of the design (eg. the functional decomposition), the *Common*KADS approach for analysis is still valid. Eg. the Fraudwatch application [Killin, 1993], designed for detecting credit cards frauds, has been reused for the design of the TRAP system which detects potential frauds in share applications [Porter, 1992].

- efficiency:

  Pure inference layer reasoning involves two mappings, which both have to be done during reasoning.

  Mixed inference/domain layer reasoning involves more transformation during design, but can be more efficient during reasoning. Eg. it can be very expensive to map a complete domain layer knowledge base which is used in a static knowledge role to a form suitable for a constraint satisfaction algorithm and simulate the workings of the constraint satisfaction algorithm on the inference layer.

  Mixed inference/domain layer reasoning still transforms data to the inference layer. So pure domain layer reasoning is the most efficient solution when a lot of data is passed between inferences. No inference layer specific mapping is involved during reasoning neither for data or for static knowledge roles. Eg. it makes no sense to transform whole databases to the inference layer during reasoning.

- maintainability:

  When we look at maintainability we restrict ourselves to those circumstances in which knowledge and data is changed, but where the overall inference structure can be kept intact. If the inference structure itself would change, the considerations mentioned for reusability are applicable because parts of the overall knowledge base have to be redesigned.

  - Pure inference layer reasoning and mixed inference/domain layer reasoning:

    When knowledge or data is changed at the domain layer, nothing has to be changed at the inference layer. Role mappings often can be held intact, but sometimes have to be changed when domain layer sorts are changed [6].

  - Pure domain layer reasoning:

    The former solutions have in common that the input for a method is indirect via a mapping. In pure domain layer reasoning the method is written is such a way that the mapping get its data via parameters. So instead of adapting the mapping, the method (or the parameter specification in the method) has to be rewritten.

- explainability:

  We can make a distinction between explanation in terms of inference layer constructs, (**inference layer explanation**), explanation of the domain layer reasoning in terms of the CML domain layer (**domain layer explanation**) and explanation about the reasoning in terms of the inference method computation (**method explanation**). We assume that the latter is only defined on high level inference methods (eg. a subsumption algorithm) or gives an abstracted view on the method (eg. a sort method could explain why its sort criterion holds for certain pairs, but it should say nothing about its sort algorithm).

  Domain layer explanation involves a mapping from method specific representation to method independent CML-type domain layer representation. This would involve the simulation of the whole reasoning inside method in terms of the CML domain layer. Eg. a procedure which catches the trace of a Prolog interpreter and reformulates this trace in terms of the CML.

  - Pure inference layer reasoning:

    For inference layer explanation the inference method is like a black box and explanation facilities only have to be designed for the inference procedures in

[6] In some applications role mappings are not only defined on sorts but also on specific instances of sorts. For example the usage of the value of an observation, which is used as a red flag to trigger an alternative inference layer step. One could argue, that red flags should have their own sorts on the domain layer. This would enhance reusability on the inference layer, although it makes the domain layer representation more inference specific.

terms of inputs and outputs of inference methods. Domain layer explanation is not applicable, because no reasoning on the domain layer is involved.

– Mixed domain/inference layer reasoning:

  All types of explanation can be used.

– Pure domain layer reasoning:

  Inference layer explanation can not be done directly, but somehow the communication between inference methods have to be traced and their contents have to be reformulated for explanation (figure 4.6).

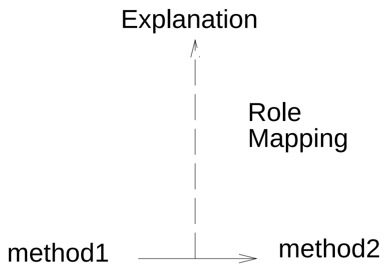


Figure 4.6: Inference layer explanation for pure domain layer reasoning

Pure domain layer reasoning of course does not have consequences for method explanation.

4.3.4 Specification of inference procedures

The aforementioned considerations about types of parameters, roles mappings, methods and how they are used inside inference procedures, gives the following entries in the specification of a method:

inference-procedure-spec	::=	**inference procedure**	*name*
		parameter-definitions	
		layer	layer-specification
		⟨inference-method⟩⋆	
		specification	*(pseudo)code*
		end procedure	
inference-method	::=	**method**	*name*
		layer	layer-specification
layer-specification	::=	⟨**inference layer** |	
		domain layer |	
		inference-domain layer⟩	

4.3.5 Some examples of inference procedure specifications

The following examples contain the HADL specifications of some design decisions which were made for the operationalisation of an $(ML)^2$ specification of a scheduling domain.

The complete $(ML)^2$ specification can be found in [Balder *et al.*, 1992] and its operationalisation in [Duursma *et al.*, 1992]. The complete design model of this application is documented in [Terpstra, 1993].

In short, scheduling is conceived in this model as making an schedule which consists of a set of assignments of consumers to providers (resources). First a set of candidate assignments are generated (inference generate). The assignment which is the most optimal wrt. user provided preference criteria, is selected (selectAssignment) and added to the schedule. The new (partial) schedule is evaluated on whether it does not violate some user provided constraints and whether it is complete (evaluateSchedule)[7].

The HADL specifications of these inferences illustrate amongst others the following design decisions

- Generate is an example where we deviate from the structure of the expertise model. In stead of one inference procedure, we used two. The two operationalisations are an example of pure inference layer reasoning in the sense as defined in the former section.

- selectAssignment contains an example of a method, which uses mixed domain/inference layer reasoning. The input for the method is a set of inference layer assignments. The method maximal element compares them on the domain layer.

- evaluateSchedule is an example of an inference, which is operationalised by two methods. The first one works on the domain layer. The second one on the inference layer.

Inference generate

The PIA generate is defined in [Balder *et al.*, 1992] as follows.

[7]We have changed some names to improve comprehensibility

theory generate
  use consumers, providers, objectDescription, culpritsPlusContext, assig nments ;
  signature
    predicates
      already–assigned : consumer × assigned set ;
      known–as–violation : assigned × culpritContext set × ass igned set ;
      PIA–generate : consumer set × provider set × assigned s et ×
          culpritContext set × assigned set ;
  axioms
    PIA–generate(Consumers, Providers, ObjDescr, Culprits, Assignments) ←
      Consumers = {C | $\text{input}_{\mathsf{consumers}}$(C)} ∧
      Providers = {P | $\text{input}_{\mathsf{providers}}$(P)} ∧
      $\text{input}_{\mathsf{objectDescription}}$(ObjDescr) ∧
      $\text{input}_{\mathsf{culpritsPlusContext}}$(Culprits) ∧
      Assignments = {assignment(C, P)|C∈Consumers ∧P∈Providers ∧
          ¬already–assigned(C, ObjDescr) ∧
          ¬known–as–violation(assignment(C, P), Culprits, ObjDescr)};

    already–assigned(Consumer, ObjDescr) ↔
      ∃ Provider : provider assignment(Consumer, Provider) ∈ ObjDescr;

    known–as–violation(Assignment, Culprits, ObjDescr) ↔
      ∃ $\mathsf{ObjDescr}_1$: assigned set
        $\mathsf{ObjDescr}_1$ ⊆ (ObjDescr ∪ {Assignment }) ∧
        culprit($\mathsf{ObjDescr}_1$, Assignment) ∈ Culprits;
end–theory

The argument objDescr denotes the partial schedule, computed in previous loops. The argument culprits denotes the set of assignments, computed so far and which do not satisfy the requirements as defined by the DKR requirements. The PIA generate generates a set of assignments, consisting of pairs (consumer,provider), which which are not already assigned to each other and which are known not to be culprits.

Generate has been operationalised by two inference procedures init assignments and generateAssignments. Init assignments is an example where we did not follow the $(ML)^2$ specification. All consumers and providers are lifted once during the initialisation of the task for efficiency reasons. In the $(ML)^2$ specification they were lifted each time when generate was invoked.

inference procedure *init assignments*
  operationalises generate
  input from domain layer
    consumers **type** set of atoms
        role type dynamic **from** caseData
        mapping consumersMapping **formalism** rewrite grammar
        direction up **exhaustivenes** all
    providers **type** set of atoms
        role type dynamic **from** caseData
        mapping providersMapping **formalism** rewrite grammar
        direction up **exhaustivenes** all
  parameter output

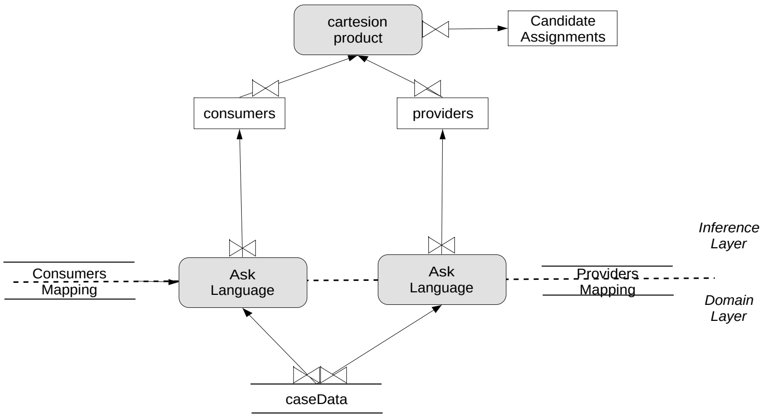


Figure 4.7: InitAssignments: Pure Inference layer reasoning

candidateAssignments **type** set of pairs(atom,atom)
layer inference layer
method cartesionProduct: set x set **layer** inference layer
spec
candidateAssignments ::= cartesionProduct(consumers,providers)
end procedure

Generate Assignments gets a set of initial assignment candidates and filters this set on assignments which contain a consumer, which is already assigned in the schedule or on assignments, which are known to be inconsistent wrt. the current schedule (culprits).

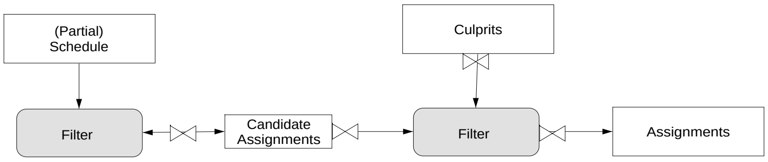


Figure 4.8: Generate Assignments: Pure inference layer reasoning

inference procedure *generateAssignments*
operationalises generate
parameter input
candidateAssignments **type** set of pairs(atom,atom)
Schedule **type** set of pairs(atom,atom)
Culprits **type** set of pairs(atom,atom)
assignments **type** set of pairs(atom,atom)
local variables
Ass **type** pair(atom,atom)
parameter output
assignments **type** set of pairs(atom,atom)
layer inference
method filter **layer** inference layer
spec
assignments ::= (filter candidateAssignments
:test knownAsViolation(Ass, Culprits, Schedule)

(filter :test alreadyAssigned(Ass ,Schedule)
candidate-assignment))

end procedure

Inference Select Assignment

theory selectAssignment
use assignments, selectionHeuristics ; assignment ;
signature
predicates
other–preferred : assigned set × assigned ;
PIA–selectAssignment : assigned set × assigned ;
axioms
PIA–selectAssignment(AssS, Ass) ←
$\text{input}_{\mathsf{assignments}}$(AssS) ∧
assignment(A_1, T_1) ∈ AssS ∧
∀A_2 : consumer ∀ T_2 : provider
assignment(A_2, T_2) ∈ AssS →¬$\text{ask}^{\vdash}$(caseData, preference(T_2,T_1));
end–theory

Select assignment selects the most preferred assignment of a list of assignments. We can identify two methods which are needed for such a selection: (1) an algorithm which implements an optimal way to enumerate pairs of assignments to be compared and returns the overall "best" assignment and (2) an algorithm which compares two assignments and decides which is the most preferred.

The method maximal element passes through the list of assignments and constructs pairs for comparison. Maximal element is an inference layer method: the data it knows about are inference layer terms (assignments).

Preference of assignments is defined on the domain layer in terms of Horn Clauses. Therefore the computation of preference is done at the domain layer by the "method" SLD resolution. Otherwise the whole domain layer would have to be mapped on inference layer terms and the inference method sld-resolution would have to be simulated in an inference structure.

Thus the inference procedure select assignment is an example of mixed domain/inference layer reasoning.

inference procedure *selectAssignment*
parameter input
assignments **type** set of pairs(atom,atom)
input from domain layer
preference(X,Y) **type** function assignment x assignment →boolean
role type static **from** caseData
mapping selectionHeuristics **formalism** rewrite grammar
direction up and down **exhaustivenes** one
local variables
Ass1,Ass2 **type** pair(atom,atom)
parameter output
assignment **type** pair(atom,atom)
layer mixed-domain-inference
method SLD resolution query **layer** domain layer

method maximal-element: set x test → element **layer** inference layer
spec
assignment := maximal-element(assignments
:test **call-on-domain-layer**(SLD query,
map-down(preference(Ass1 ,Ass2)),
caseData))
end procedure

Evaluate schedule uses two methods. First the schedule is checked whether it does not violate user defined constraints. This is done by the method satisfyRequirements on the domain layer. If there are no violations, the method setCompleteness looks if the schedule is finished, which is defined in a generic way: are all consumers assigned in the schedule. setCompleteness works on the inference layer.

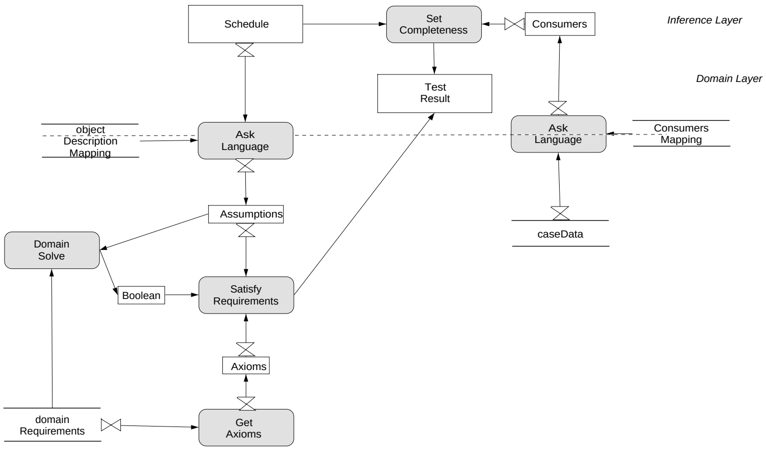


Figure 4.9: Evaluating the schedule.

inference procedure *evaluateschedule*
parameter input
schedule **type** set of pairs(atom, atom)
input from domain layer
consumers **type** set of atom
role type dynamic **from** caseData
mapping consumerMapping **formalism** rewrite grammar
direction up **exhaustivenes** all
schedule **type** set of pairs(atom, atom)
role type dynamic **from** requirements
mapping scheduleMapping **formalism** rewrite grammar
direction down **exhaustivenes** all
requirements **type** Horn clauses
role type static **from** domainRequirements
mapping getRequirements **formalism** method specific
direction none **exhaustivenes** all
local variables
consumer1 **type** atom
parameter output
testResult **type** atom

```
method setCompleteness: set x test → boolean layer inference layer
method satisfyRequirements: set of assignments x set of horn clauses →bool layer domain layer
spec
    if not call-on-domain-layer(statisfyRequirements(map-down(schedule), domainRequirements))
    then testResult := violation
    else if setCompleteness(consumers, :test member(consumer1,schedule)
        then testResult := finished
end procedure
```

4.4 Summary

The subject of this chapter is one of the possible design routes for application design: structure preserving design. In structure preserving design, the expertise model is used as a functional specification of (a part of) the final knowledge based system artifact. Application design is then the process of refining the initial knowledge level specification of expertise model components in such a way that they can be realised in the computational architecture as provided by the architecture design.

A goal of structure preserving design is to preserve as much as possible the structure of the expertise model in the final artifact. Wrt. the design product this has obvious advantages for maintainability, reusability and explainability. Wrt. the design process an advantage of structure preserving design is that it decomposes the design process into a set of manageable design decisions like

- selecting control regimes for the task layer,
- selecting computational methods for inferences,
- selecting mechanisms for role mappings (figure 4.10) ,
- defining the locus of action of inference methods (figure 4.10) ,
- selecting appropriate knowledge representation schemes for the domain layer.

Moreover this chapter shows that such a decomposition makes it possible to give quite detailed guidelines how to choose between different operationalisation options, by identifying what their impact is on quality criteria like efficiency, reusability, maintainability and explainability (risks).

In this chapter we have developed a notation, (High level Application Design Language, in short HADL) to document the aforementioned design decisions. Each identified design decision has got a slot in the HADL. The resulting notation can be used for mapping expertise models on a wide range of architectures.

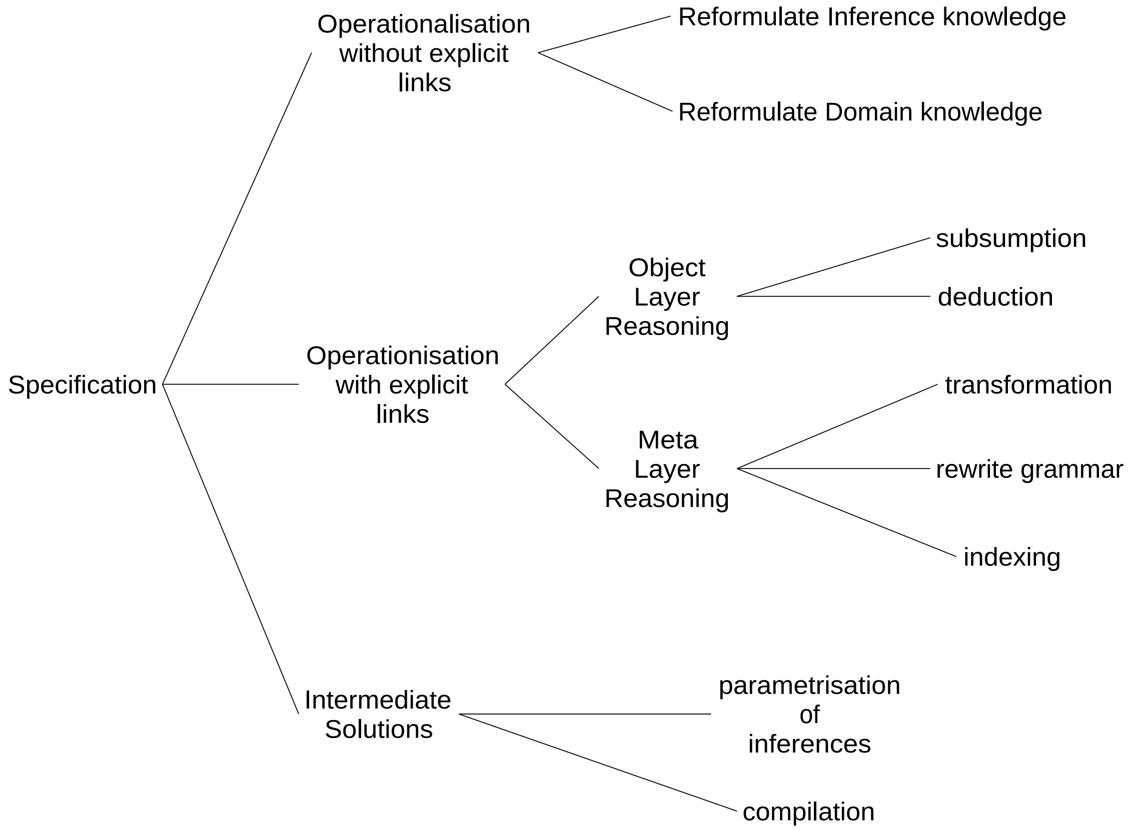


Figure 4.10: Summary of different operationalisations of the relation between domain and inference layer

Chapter 5

Design Library framework

Olle Olsson

This chapter is dedicated to the topic of a design library. By a design library we mean a repository containing any kind of information that aids the designer during the design process.

In the first section we describe different types of design decisions, and the kinds of knowledge that can support these decisions.

In the succeeding section, we will describe what kinds of information the design library can contain, information that supports the design decisions.

5.1 Design decisions and the design process

This section provides a framework for understanding the role of design decisions in the design process.

First a classification of design decisions will be given.

Then factors influencing design decisions will be presented.

5.1.1 Basic design decisions

When elaborating a design, there are typically a number of different ways in which the design model can be extended into more detailed designs. We can say that when the design model is in a certain stage, there are some possible ways of extending the design into more detailed descriptions.

When there are different design routes available, the designer chooses one for elaboration, typically corresponding to a certain part or aspect of the design. Then he proceeds in performing the elaboration.

This corresponds to two types of decision making in the design context:

planning design: deciding on *what part* of the design to elaborate

performing design: deciding on *how* to elaborate it.

The second type of decisions, performing design, is technical decision making, extending the design by adding more technical details. The rationale for making such a decision is typically based on purely technical reasons.

A decision of the first type, planning design, can be based on purely technical reasons, or it can be based on non-technical reasons. An example of the former kind is when the designer decides that the technical viability of the entire design is dependent on whether a certain part of the design can fulfill the technical constraints imposed by other parts of the design, and therefore decides that this part should be elaborated.

An example of the latter kind of decision is when there is a question about whether the requirements on the application can be fulfilled by the present design, and the answer to this question can be illuminated by elaborating on a certain part of the design.

From the perspective of how the design process is to be managed within a CommonKADS development process, we see that decisions concerning planning of the design should be conducted in a risk-driven, quality-oriented way. The design process should take a route that highlights or minimises the risks involved, and that maximises the quality attributes of the resulting design. As we will show later, the design library can provide support for planning, by offering advice on quality and risks in making certain design decisions.

Whether decisions concerning technical elaborations can have an impact on the risks and quality attributes depends heavily on the technical context of the current state of the design model.

To clarify the problems involved, we need to distinguish between different kinds of constraints on the design model. One type of constraints comes from the any of the organisation model, task model, agent model, communication model, and expertise model (we will use the term *specification model* to stand for any of these five models). The constraints coming from the specifications models will be called *requirements constraints*. Another type of constraints originate in earlier made design decisions. These will be called *design constraints*.

Design decisions resulting in a technical elaboration of the design model need to be evaluated with respect to both requirements constraints and design constraints.

At the beginning of the design process, there are only requirements constraints, as no design decisions have yet been made.

As the design process proceeds, the set of not-yet satisfied requirements constraints will shrink, and the set of design constraints will increase. This means that in the early phases of the design process, it is more important to evaluate technical design decisions with respect to their impact on the quality and risks correlated to the requirements constraints. In the later phases of the design process, it will become more important to make technical design decisions that satisfy the design constraints.

There is yet another type of design act; design evaluation. Evaluation is performed to investigate whether the current design model still fulfills the set of requirements constraints and design constraints. It will also provide a better foundation for understanding the risks involved, and the effects on the quality criteria.

5.1.2 Factors influencing design decisions

We have earlier distinguished between functional and non-functional requirements. We will now introduce an orthogonal classification of requirements, and describe how it can support the design decision making.

Certain kinds of qualitative requirements in the specification models can be identified and labelled, because they are of general utility when making design decisions. Such requirements will be called *specification features.*

Some specification features can be derived from task features, but in general, specification features constitute a wider class.

The following are some examples of specification features, and in what way they influence the design process:

- support explanation of problem-solving behavior

  This will exclude some architectures or design paradigms from consideration, and will include others for consideration.

- interrupt-and-resume the problem-solving process.

  A requirement may be that an agent is allowed to interrupt an ongoing problem-solving process, start or re-activate another problem- solving process, and then at a later time resume the interrupted process.

  This will have effects on what architecture to choose.

- robustness with respect to unreliable input.

  The fact that input data cannot be trusted might either have been tackled in the specification models (i.e., the way to handle unreliable input has been modelled), in which case it does not influence the design process. There can be several reason why this kind of modelling has not been done in the specification models. E.g., that this was detracting from the real problem; that it required the modelling of very complex sensor devices; that the sensors will be replaced by better ones, but in the meantime the system must have some kind of robustness.

  If ways of handling unreliable input was not modelled, then it must be handled at the design level.

- flexibility in the initiative during problem-solving.

  A requirement might be that the user should be able to take over any kind of problem-solving activity, and give the result back to the system. This constitutes a strong requirement on the architecture supporting the problem-solving of the system.

As this list indicates, qualitative specification features may influence major design decisions during the design process.

Specification features can be used in two ways:

- providing guidance on making a *constructive* design decision
- providing guidance in *evaluating* the state of the design model.

In both of these roles, specification features can be encoded in guidelines, either recommending certain types of technical solutions to satisfy certain requirements, or informing the designer that the current state of the design is not consistent with the requirements.

5.2 Design library

The purpose of a design library is to support the design process, achieving the following kinds of effects:

- accelerating the design process, allowing the designer to make leaps in the design process by incorporating ready-made parts of designs
- decreasing the amount of back-tracking in the design process, by focussing the designer in a good direction
- increasing the quality of the resulting design, by re-use of validated partial designs

These effects will be achieved if the different types of design decisions that we have mentioned in the previous section can be made more efficient.

There are well-known problems in creating an environment that stimulates development based on reuse. As described in [Hemman, 1992] these problems can be grouped into technical issues (design approaches, languages, tools, etc) and social issues (organization, standardisation, etc). This means that the existence of a design library itself is not a sufficient prerequisite for achieving the goals mentioned above, but it must be regarded as a necessary prerequisite.

Using a design library involves three different activities:

- searching the library for useful information
- evaluating and selecting the information to use
- taking advantage of the selected information in the context of the current design

One type of information that can profitably be stored in a design library is partial designs or parts of designs. This type of information will be called *design components*.

A design library can contain any kind of design components, as illustrated in the following list:

- general architectural principles, e.g., a blackboard architecture.

  This can be represented in the library as a frame indicating the properties of the architecture and indicating the kind of design decision that need to be made if this architecture is chosen.

- design structure fragments, e.g., a design for representing and accessing information stored as a semantic network.

  Such design library structures can be represented as a design skeleton, with specific slots to be filled, to match the needs of the current design.

- particular pieces of code, e.g., the implementation of a searching algorithm, traversing a hierarchic data structure.

  These parts of code can be self-contained, or they may have some fillers that need to be specified for the current design.

- particular software systems, e.g., a commercial rule-based shell.

  These systems are not stored as such in the library, only a description of the system is kept there.

As illustrated above, library components that support design range in granularity from architectural principles to actual code.

For a library to be useful, it must be possible to retrieve information from it, information that is appropriate for a certain design decision. Information that can be used as indexes into the design library are called *design indicators*. A design indicator is either a property of the current design, a property that we want the next stage of the design to have, or a specification feature.These indicators are used to navigate in the library towards a library component that might be appropriate in the current design situation.

A design component can, in addition to design and code structures, contain design indicators that makes certain properties of the component explicit. If the component is accepted by the designer and incorporated into the design model, the design indicators of that component will add to the characterization of the new state of the design model.

The main use of the library will be to support the technical elaboration of the design model. Hence it is mainly the technically oriented design decisions that will profit from using a design library. This does not prevent the library from containing information that is of relevance for non-technical decisions. A design component might be annotated with information that can be useful for risk- or quality-oriented evaluation of the design.

Example 1: A design component contains the design of a procedure that searches for those elements in a very large data structure that fulfill a certain criterion. The design might be optimised for execution speed, and it cannot guarantee that the elements will be returned in any specific order. This design component should be annotated with the following information "The component cannot guarantee a specific order of returned elements". This un-order property can propagate to other parts of the design, and ultimately to a user. It may then be appropriate to evaluate whether this non-order will present problems to the user. If there is a strict requirement that the elements should be ordered, and a strict requirement on the response time of the system, and we seem to be unable to satisfy both of them, then only a nontechnical decision can resolve the situation, either by accepting unordered element presentation, or by accepting longer response times, or by deciding that the project should be terminated.

There are two types of technical design decisions; constructive design decisions and technical design evaluation decisions. Both of these can be supported by a library. We earlier gave some examples illustrating the library as a repository of design or implementation components. But as the design indicators used in navigating through the library can express both technical aspects and requirements-oriented aspects (through the specification features), it is possible to retrieve any kind of information that can be indexed through such design indicators.

Example 2: if a decision was made to use a certain commercial forward-chaining rule-based expert system shell, and it was later decided or detected that explanations must be presented to the user on demand, this latter specification feature might be in conflict with the design indicator that the current design inherited from the expert system shell. Hence the current design will not be able to support the explanation requirements on the application to be built.

Example 3: if there is one known specification feature stating that working memory storage is limited, and another specification feature stating that explanations need to be provided, the conjunction of these can be used as design indicators searching the library for a suitable architecture. The library may then return a hit which is not an architecture at all, but only contains a message (which we call *design advice*) of the following form: "The requirement on memory size and the requirement on explanations may be in conflict with each other".

All the examples above show that a library can support design evaluation decisions (both technical and non-technical) as well as constructive design decisions.

As Example 3 illustrates, the library should return some kind of information for a search. If there is an inherent semantical conflict between design indicators, the user should be informed (in terms of design advice) about this conflict, instead of getting no hit in the library.

By keeping design advice and design components (i.e. constructive design recommendations) on an equal status in the library, we get a library that can be evolved in two directions; by adding to the set of components in the library, and by adding more design advice.

In the following section we will present how design components are represented in the library.

In the section after that, we will present the indexing structures, which supports the user in his search for suitable design library elements.

It is important to note that the descriptions that follows are at a conceptual level. They are not to be interpreted as a functional specification of a design library support system, rather as an incomplete characterization of what a design support library should provide.

5.2.1 Design Library Component Representation

There are a number of approaches to how a library should be structured, heavily dependent on (1) what kind of entities the library contains, and (2) for what purpose the library is to be used (what design tasks the library supports). In the context of a library intending to support development and maintenance of some specific large software system certain approaches are useful, as illustrated in LaSSIE [Devanbu et al., 1991]. In the context of more general software development, other approaches seem viable.

An approach that has been shown to be useful is to characterise library components in a number of *facets*, corresponding to a number of perspectives of the components [Prieto-Diaz and Freeman, 1987] [Prieto-Diaz, 1991]. We will adopt this general way of characterising library components. This section presents a facet structure of the library design components, facets that allows useful properties to be represented.

Please note that we are not committed to any specific implementation representation. How design components can be represented in an actual library management system will be dependent on the features of that system. The proposal here is to be regarded as a set of desirable requirements. It is perfectly possible to imagine a paper-based design component library.

The following list describes the different facets (or slots) that should annotate design components.

Component name: This slot contains the name that uniquely identifies the component.

Component: This is where the design component itself is represented, in any kind of form that is useful. As the Design Model provides a structural design representation, this representation should be used.

Design component type: This slot contains the type of the component, according to the constituents defined in the design model.

Examples: This slot can contain values such as

```
subsystem
detailed-application-design
computational-method
....
```

Design decisions: This components may involve a number of design decisions. This is especially true for partial or skeletal components of large granularity. Such a components is the result of a certain number of design decisions, and incorporating this component into the current design implies that design decisions made in the library component are inherited by the current design. This slot is used to express all such already made design decisions.

As there is a specific CommonKADS application design notation (see the chapter on Design Model Notations), it is recommended that this notation is used whenever possible.

CommonKADS type: This slot is used to annotate the component with those terms, used in the CommonKADS methodology, that best describes what this component can be used for. This provides a link to the context of the specification models, where these terms are used for typing and characterising the elements.

Examples:

```
transaction
inference:select
task-feature:collection-of-components
...
```

Description: A high-level natural language description of this component. It should include both functional and non-functional characterisations that are not easily described by other properties of the design component. Furthermore, important remarks to the designer can be added here, e.g. critical properties of this component that needs to e verified as acceptable in the current design.

Reference: Pointers to more elaborate descriptions of this component, to theoretical underpinnings of the component, or to any other kind of source that can give more information about this component.

Example:

```
See: AI Journal, vol 22, p. 122-157
```

Function: For objects that exhibit of a functional or procedural behavior, this slot is used to characterize the function in terms of a number of pre-defined categories.

Examples:

```
search-tree
theorem-prover
forward-chaining-rule-evaluator
....
```

Input: When the component uses some input, a characterization of the type of input can be given.

Examples:

```
directed-graph x element-pair
set
....
```

Output: When the component produces some output, a characterization of the type of output can be given.

Examples:

```
element
set
list
....
```

Required components: The design component may require that other design components from the library should be present in the design.

Examples: A skeleton application design may be expressed in terms of a certain computational methods. For the application design to be used, these computational methods need to be present in the design. If these specific computational methods are provided by a specific architecture that is present in the library, that architecture should also be integrated into the design.

Execution-time properties: KBS applications usually involve complex searching and matching, and procedures for these can exhibit quite complex behaviors. Execution-time can be a critical factor for the viability of the design, and early knowledge about the execution-time properties of the corresponding procedures is useful as a selection criterion during design.

Examples:

```
Execution-time is exponential in the size of input data
```

Execution-space properties: Analogous with the preceding property, procedures for KBS sometimes uses a lot of working space during computation. If memory space is a restricting factor on the hardware/software platform being envisioned, then the need for working space can be a good selection criterion for design component selection.

Examples:

```
Working-space needed is linear in the size of input data
```

Other exhibited component properties: The two items above describe properties that characterise this design component. There are of course un unlimited number of possible properties. Some of these properties can be of importance for judging the usefulness of this component in a certain design context. As it is not possible to enumerate all potentially interesting properties, this slot provides for the representation of an extensible list of properties. They come in pairs, of the form <property, value>, where a value can be of any kind (number, text, boolean,...).

Examples: A blackboard-based inference engine might find the following annotation:

```
<interrupt-resume-inferencing, true>
```

meaning that a certain inference process can be interrupted and after doing some other computation, the inference process can be resumed.

An application architecture object can be annotated with

```
<provides_explanations, false>
```

meaning that it does not provide any mechanisms for producing explanations.

As is seen from the last example, it is possible to express negative properties. This is quite useful, because it definitely excludes a component from consideration if the property is inconsistent with what is required. In other words, these kinds of properties have a three-valued semantics, as is seen from the following example:

```
requirement:    <provides_explanation, true>

Component 1:    <provides_explanation, true>

Component 2:    <provides_explanation, false>

Component 3:
```

Component 1 satisfies the requirement, Component 2 fails the requirements, and Component 3 is undecided. In this way we avoid the problems of a closed-world assumption, as CWA is only valid in domains where we have complete knowledge. In a design library we cannot trust that all components will be annotated completely, and we do not know what properties a designer will use in a query to the library.

Required design properties: The slot described above makes a definite statement about the component itself. In some cases it is necessary to require certain properties of the design being elaborated, if this component is to be used. As above, the value is a set of <`property, value`> pairs.

Examples: An interaction architecture may be characterized by

```
<requires_color_screen, true>
```

which says that it is capable of using different colouring schemes in the interaction, and for this to work properly, the design has to use a platform providing interaction in colours.

5.2.2 Design Advice Representation

As motivated earlier, when using the library not only components should be returned to the user, but also other kinds of non-component information. We called the latter *design advice*. The aim is to store guidelines, recommendations, warnings, and other kinds of informative stuff in the library, and when the user searches for something in the library, such information might be returned instead of a component.

As we see the design advice as something belonging to a library, we need a representation, in which we can store whatever we regard as design advice.

For regularity reasons, we reuse the same representation as for design components. Of course some slots will not be appropriate, e.g. the Component slot where the actual component is stored for real design component objects.

The actual design advice that should be made available to the user will be stored in the Description slot.

Having this uniform representation of all library elements, will enable uniform search and access mechanisms to be used for both advice and components. In this way, the user will be given some piece of design advice, if the search for components also matched this advice. I.e. advice will be available when it is appropriate.

When we in the next section describe searching in the library, we will do that in terms of components, but the reader should remember that everything stated there is also applicable to design advice.

5.2.3 Design Library Indexes

As we have assumed a facet-oriented characterisation of the components of the library, it is obvious that these facet characterisations will be a basis for locating components. Several techniques for matching queries against a set of components have been devised. E.g. in [Prieto-Diaz and Freeman, 1987] a technique of fuzzy matching based on metrics was defined. We propose a somewhat simplified approach, by providing indexes in the form of structured *term spaces*. This has been used as an important part of the REBOOT approach to reuse [Thunem and Sindre, 1992] [Sorumgard et al., 1992], where four facets are used (abstraction, dependencies, operates-on, operations), and for each of these facets structured termspaces are defined.

This section will provide a description of the different term spaces (indexing structures) that supports context-dependent directed search in the library. These term spaces are hierarchically structured, along a dimension from more specific terms to more abstract terms (specialization/generalization, cf [Sorumgard et al., 1992] p. 47, where a hierarchical termspace for operations is shown).

It is important to emphasize that we are not giving a specification of the library or of its user interface. We will not be committed to what kinds of interfaces are available, and in what form the questions are posed and the answers are presented. We can characterise this as a kind of CommonKADS task model description of the library search task. I.e. we are interested in describing the *kinds* of information that is used in searching in a library, not their exact representation (as would be described in the CommonKADS Communication Model).

Therefore this discussion relates to the specification of the query functionality of the generic library facility (see section 5.7 of [Van de Velde, 1993]).

We will describe the interaction between the user and the library as a question-answer interaction. The user poses questions, and the library provides the answer. We can differentiate between different kinds of questions, depending on the degree of detail in the question.

In certain situations, the user knows quite well what he wants. He is well acquainted with the contents of the library, and he would like to pick up a component which he knows is there. This results in the question being exact, and the library user should be able to specify in complete detail the search criterion that locates the component in the library.

In other situations the user does not know exactly what he is looking for, and hence cannot compose a detailed question for the library. He would like to access the library and get information that will provide help of two kinds:

- help in knowing how to pose a more exact question. This is the problem of mapping semantics (what he already knows about the design) to syntax (how to phrase this knowledge in terms of questions)
- help in understanding what knowledge is needed in order to pose a good question. This is about the problem of acquiring new knowledge about the design, or of making implicit knowledge explicit

The well-defined and structured situation and the un-structured situation are the two extremes that delimit the spectrum of uses of the library. We cannot assume that one indexing structure can support all these kinds of situations, and consequently we need several indexes, corresponding to several dimensions of search.

The following indexes are defined:

Design component type index: this is a index that provides direct access to the set of components of a specific design component type. Posing a question in terms of this index will return the set of components of the specified type.

The index has only one level, the branching to the different component types.

Examples:

```
application-design
computational-method
...
```

CommonKADS type index: Components can be annotated in their CommonKADS Type slot with appropriate terms from the CommonKADS methodology. As there is a structure among these terms, it is useful to be able to represent that structure explicitly. This will help in locating components when starting design from the specification models.

Examples:

```
inference
   inference:select
   inference:match
   inference:filter
task-feature
   task-feature:goal
      task-feature:goal:collection-of-components
      task-feature:goal:explanation
   task-feature:output
      task-feature:output:binary
....
```

Function index: this indexing structure is in terms of functional characterisations. We can connect design components to a specialization hierarchy where the nodes are functional descriptions. Traversing a link from a node to a subnode corresponds to selecting a more specialized functionality.

As the set of all kinds of functionality can be partitioned in disjoint subsets, corresponding to the the different design component types, this index provides a hierarchy, and all components that are referred to by this index will all be of the same component type.

Examples:

```
search
  search-linear
  search-tree
     breadth-first-tree
     depth-first-tree
  search-graph
match
  match-linear
  match-tree
  match-graph
...
```

Input data index/Output data index: Both of these are hierarchical indexes of data types, where a subnode to a node describes a more specialised version of the data type of its supernode. They can be used as paths to locate components having input or output matching a specified type.

Examples:

```
any
   set
      set-with-duplicates
      set-with-no-duplicates
   list
      pair
   tree
   graph
      undirected-graph
      directed-graph
        acyclic-graph
        cyclic-graph
...
```

These indexes have an existence that is independent of the components of the library. This means that the indexes are "in" the library even if the library contains no components at all. This separate existence is present because the indexes express knowledge about concepts that can characterise components. E.g. the Input/Output Data index contains classification knowledge, and the CommonKADS Type index expresses knowledge about structure among the terms used by CommonKADS. These indexes provide one source of knowledge that can help a designer to sharpen his understanding of his application, so that he will be better equipped to pose detailed an unambiguous questions to the library.

Searching the library for components can be done by traversing the indexes identified above, by doing a global search for design library components having slots matching an explicit search criterion, or by a combinations of these two mechanisms.

Traversing the indexes is equivalent to constructing a query corresponding to the path that is traversed. The main purpose in using the indexes in this way is that the user is informed about the options being available, about how he can construct a more focussed query.

This means that if a component is found after traversing some indexes, the same component could be retrieved by a complete query, and vice versa. In other words, with respect to locating components, index traversal and query formulation have equal expressibility.

Explicit search criteria are typically expressed in terms of the following slots of the design library components:

```
design-component-type
commonKADS-type
function
input
output
execution-time properties
execution-space properties
other-component-properties
```

These are the slots that seem to be most useful, from the perspective of what is known about the current state of the design. There is, however, no restriction in using other slots as search criteria, but it does not seem to be useful to make a search for the `reference` slot, for example.

It should be noted that the matching process is not restricted to exact matches only. As most indexes have a hierarchical structure representing specialisation hierarchies, a component matches the entire path from the root of the tree to the the most specialised match.

Example An architecture component X in the library has as value in the Function slot `forward-chaining-rule-evaluator`. In the Function Index we have the following structure:

```
any                                                (0)
   search
   ...
   inference-engine                                (1)
      theorem-prover
      ...
      rule-evaluator                               (2)
         forward-chaining-rule-evaluator           (3)
         backward-chaining-rule-evaluator
         ...
      ...
   ...
```

Component X will match exactly the function characteristic noted as (3), but it will also match (2), (1), and (0), as component X also can be regarded as a `rule-evaluator`, as an `inference-engine`, and as `any`.

The following figure describes the multidimensional space of indexes in which the design components reside:

5.3 Summary of this chapter

The aim of this chapter was to present a structure of the library of design components. To provide a good motivation for the structure proposed, we characterised the set of design decisions that a designer makes during the design process, and identified the kind of support that a library should offer to the user.

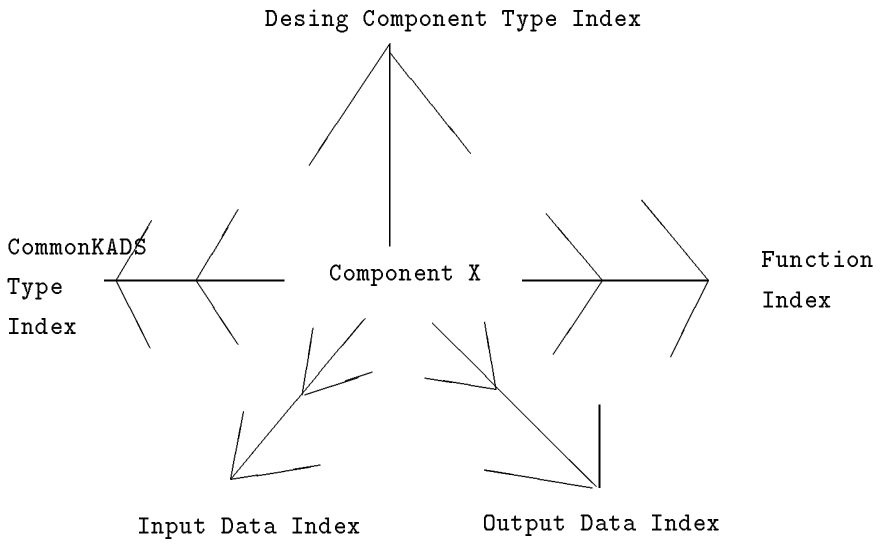


Figure 5.1: The multi-dimensional library index space

Part II

Design Approaches

Chapter 6

Relationships with and Dependencies on other Models

Olle Olsson, Ulf Sundin, Maria Gustavsson

The topic of this chapter is to present an illumination of the different ways in which the Design Model depends on the other CommonKADS models, and in what ways these dependencies influence the progress of the the design process.

The first section explicates on the concepts of specification and design. What is a specification, and what is a design?

The second section describes in some detail how different models provide restrictions on the result of the design process, i.e. restrictions on the design artifact.

6.1 Specification and Design in the CommmonKADS Context

In a CommonKADS development project the set of models to be developed is configured into a model configuration. During the lifetime of the project this configuration may be modified, reflecting project management's awareness of the risks involved, and reflecting the quality standards set for the project.

We can expect that if and when work on the design model is initiated, then there is a confidence that the problem to be implemented is well understood and well described. I.e. information about the problem has been gathered, analysed, structured and recorded in those of the following five CommonKADS models that has been included in the current model configuration: organisation model, task model, agent model, communication model, and expertise model.

The characteristic that distinguishes the design model from the other CommonKADS models is that the design model should not add to the knowledge about the problem. The design model is expected to provide another viewpoint on the problem, the operationalisation viewpoint. As design work is a time-consuming and risky kind of work, it should not be based on incomplete knowledge, or on unreliable knowledge about the problem.

This means that before design work is initiated, the contents of the other models in the model configuration should be validated against the real world problem, with respect to completeness, correctness, and appropriateness, and that these models should be checked for intra- and inter-model consistency.

Given that such validation has been performed, design work can be based on as good a foundation as is deemed possible, given the risks and quality aspects on the development project.

We can also expect that models representing (validated) problem knowledge appear in two versions, corresponding to two different purposes:

- models can be used to represent knowledge about the current situation
- models can be used to represent a proposal for the future (the future situation)

A model constructed with the first purpose in mind is clearly a pure analytical model. The validity of the model is determined by comparing it against the real problem and its context.

A model constructed with the second purpose in mind will contain elements and structures that do not represent current reality, but a desirable or hypothetical future. Such a model is built on the assumption that the real world (the problem and its context) can be transformed into a state corresponding to the models. Hence such a model cannot be fully validated, as it represents a future *possible* world. Validation of information can only be performed if there is some real truths to compare it to. A reference point that can be used for validation of these models is the set of external requirements specified and agreed upon by the client of the project.

To simplify the description that follows, we introduce the term *specification model* to stand for any one of the Organisation Model, Task Model, Agent Model, Communication Model, and Expertise Model, being descriptions of the future situation. We thereby indicate that these models will provide or imply the specification of the application that will be designed.

It is important to realise that a model representing a future situation will contain elements and structures that are the result of making decisions about the future. In other words, describing the future situation is the result of making decisions about what a desirable future situation is. I.e., the future situation is *designed*.

We must distinguish between, on the one hand, the design decisions reflected in the specification models, and on the other hand the design decisions pertaining to the Design Model. In this document we will only discuss aspects on making decisions about the design model, and how such decisions can be supported by guidelines, approaches, methods, techniques, and libraries.

Designing a specification model for a hypothetical future situation can also be supported by guidelines etc, so that the future situation being defined has a high probability of being a realistically possible future situation. One example of this is the use of established principles for the design of man-machine communication, i.e. principles that will increase the probability that the communication model will represent a feasible future.

Having said that, we can now state that the design model will only reflect knowledge contained in the specification models representing the future situation.

The specification models provide restrictions on the design model, in the sense that some designs are consistent with the specification model, other designs are not. If a design is not consistent with the specification models, the design should be changed.

But, as mentioned above, the specification models express a *designed future situation* that should be consistent with external requirements on the application for the problem. The relation between the external requirements and the specification models is analogous to the relation between the specification models and the design model. It is the relation of "consistency"; the design model should be consistent with the specification models, and the specification models should be consistent with the external requirements. As "consistency" is not a functional relation, we see that the (contents of the) specification models can be modified in certain ways without becoming inconsistent with the external requirements.

This affects the design model and the design process in the following way. If the design model enters into a state from which it is impossible or unrealistic to further elaborate the design, there are two choices:

- retract some earlier made design model decision and find another way of elaborating the design model, a way that will lead to a design that is consistent with the specification models
- retract some "design decision" made in the *specification models*, to arrive at a somewhat modified specification that is either consistent with the current state of the design model, or for which one believes that there exists a possible complete and consistent design model.

A further option is to negotiate a revision of the externally enforced requirements, which may lead to a "re-design" of the specification models.

It is important to be aware of these possibilities, as real design is never a linear process that monotonically gets closer to the final design structure. Design is rather a process of exploration, where the explorations relate to the two dimensions:

- exploring the technical possibilities of the design, and of the properties exhibited of the designed artifact
- exploring the dimension of what is required of the artifact being designed, the constraints on the artifact

That the design process is an explorative process also implies that mistakes will be made, the process will progress into dead- ends, complexity will overwhelm the designer, ... An effect of this is that already made design decisions will be revoked, and new decisions will be made.

To be able to support such iterative revisions of design and specifications, it is important to provide tracability among the different pieces of information that influence decision-making. The most suitable way of providing this tracability is to provide a rationale for design decisions, annotating the decision with the related requirements expressed in the specification models.

This explorative view on the design process is entirely in consonance with the cyclic, risk-driven approach to modelling proscribed by CommonKADS. We can even make the stronger statement that the CommonKADS risk-driven approach is the best way to support explorative tasks like design.

6.2 Dependencies on other CommonKADS models

6.2.1 Application Requirements

The state of the set of specification models implies a requirement specification of the system to be designed and built. The word "implies" is used to denote the fact that the specification models may contain more information than is needed for a strict software systems requirements document.

It is also true that the state of the specification models may be too weak from a requirements point of view. I.e., they may turn out to be so incomplete that a certain design decision cannot be decided on, due to lack of information in the specification models.An example of this is where a process control system is to be built, and during design it is discovered that nowhere in the specification models is there any mention of whether the operators interact with the system using a fixed set of dedicated push-buttons, or whether they are interacting via a full-fledged terminal.

The word "requirement specification" was used in the preceding paragraphs instead of "system specification", to indicate that the CommonKADS specification models definitely supports expression of requirements on the system. By "system specification" we mean a detailed specification of the behaviour of the system, not only requirements on it. This includes some aspects that do not fit easily into the model structure of the specification models. An example of such specification is that the knowledge base should be partitioned and distributed across several nodes in a network, and that it should be accessed as a distributed data base.

There is no inherent deficiency in the specification models that prevents functional specifications to be expressible in the specification models. On the contrary. But we must expect that in practical CommonKADS development projects some functional requirements will not be explicitly expressed. This incompleteness will hopefully be discovered and corrected, resulting in more detailed requirements being expressed by the models.

We separate two classes of requirements:

functional requirements: describing required functional behaviour of the application

non-functional requirements: describing requirements that are not directly of a functional character, but that expresses restrictions on acceptable implementations of the functional behaviour.

The specification models are aimed at expressing the functional aspects (and some procedural aspects) of the application. We might even say that a total set of complete specification models *is* an expression of the functional requirements and the rationale for these requirements.

The models do not provide equal support for expressing non-functional requirements. Some of these requirements are explicitly expressed in different parts of the models, other non-functional requirements are implicitly expressed in the models.

As simple examples of non-functional requirements we mention the following:

- response-time requirements: e.g. maximum time to response, or average time to response.
- computation-space requirements: e.g. maximum working set allowed for application.
- hardware/software platform requirements: e.g. VAX, Ultrix, Oracle, NexpertObject, C++.
- peripheral equipment requirements: e.g. special operator console equipment in use.

In the following subsections, we will highlight how the different CommonKADS specification models represent functional and non-functional requirements and other constraints that influence the design decisions in the design process.

6.2.2 Organisation model

The organisation model has a strong flavour of supporting the process of *finding* and *delimiting* a problem to work on, and understanding the organisational implication of this problem and its solution. It does not, in this sense, restrict the design in any significant way. The way it influences the design is indirect, by focussing development work on the other specification models onto some dedicated problem, and it is from these latter models that the elaborated requirements can be fetched from.

There are, though, some components in the organisation model that can influence the design model.

One component is the "computing resources" available in the organisation. For many application development projects no new types of computing resources will be purchased, so the kind of equipment to use is well-known.

The available computing resources will provide constraints on the design in terms of:

- hardware platform (incl. networks)
- operating system
- DBMS and UIMS
- programming languages
- available application building tools

If we see these as organisational constraints on computing resources, then we can also expect different strength of these constraints. In the list above, the further down the list we go, the weaker the constraint. I.e. it is much easier to convince an organisation that a new application building tool is needed than to argue that a new operating system is needed.

Other requirements can be found expressed in the model constituent "knowledge", such as standards and policies concerning installation, execution and maintenance of applications. Can applications be parametrized at execution-time, or only at installation-time? Should knowledge-bases be protected against un- authorized access? Should they be encrypted? What are the general security restrictions in the organisation?

Restrictions such as these might influence the detailed design of the application and the way it is configured as an execution unit.

In the constituent "knowledge" we may also find the level of knowledge and know-how of the computing unit (data processing department) in the organisation. If there is one. For KBS systems this can be a critical issue, as it concerns the problem of keeping the knowledge base up-to-date. Who is doing application maintenance? Who should be doing it? What design decisions will influence the practical maintainability of the KBS system, seen in the context of the competence of the intended maintainers?

Summary: The Organisation Model contains some components that can be of critical importance for the ultimate success or failure of the KBS system. These components imply non- functional constraints on the design model.

6.2.3 Task Model

The task model contains much information strongly related to design. One reason for this is that the task model represents a global context of the problem being designed, a context in which the other specification models should be understood. Therefore, the task model establishes a focus for other the elaboration of other models, and offers a framework that ties together the entire analysis effort.

Another reason is that the other specification models elaborate on aspects that originate in specific parts of the task model. E.g., the communication model provides elaborations on the communication needed by tasks, and the expertise model provides a detailed analysis of a task identified in the task model.

The following components of the task entity in the task model provides (directly or indirectly) functional specification:

goal: the goal can be expressed as a transformation condition on the input and output of tasks. Hence it can be a formulation of a functional requirement

precondition: this slot can be vital for the correctness of the behavior of the application, if it expresses that starting a task execution requires more than availability of input. Preconditions are to be regarded as natural parts of a functional specification

task-control: this provides control information shedding more light on how the goals of a task can be achieved in terms of simpler functionalities

The entities of type ingredient, related to tasks as task inputs or task outputs, provides a slot:

contents: the abstract structure of the data consumed or produced by a task. This does not provide a full specification of data, as the actual formats are not specified in the ingredient, but in an associated information item entity.

Task capability entities describes the capabilities that are needed for correct performance of the task. This may provide important qualifiers to the pure task descriptions, for two reasons:

- it may indicate procedural knowledge about performing the task. Such procedural knowledge provides additional information about how tasks can be performed.

- it may identify static background information that is used in the task. Examples are tariffs, classifications, and catalogue information. This extends the description of the functionality of the task by exhibiting that the function is parametrized by a certain kind of information.

All of the model components mentioned above provide information directly related to the functionality of the task. And as design is about achieving a certain functionality, the model components are of importance for the design model.

We also find some model components that do not provide direct functional requirements, but probably will influence the design work.These model components imply non-functional requirements.

In the task entities we have slots:

frequency: denoting how often this task is performed.

performance time: denoting acceptable time from task start to task end

Both of these can be important for certain design decisions; for tasks with very high frequency and/or very short performance time, we must guarantee that the application built will respect these constraints.

Depending on the circumstances, these time-oriented constraints can influence the high-level design decisions, or may influence low-level implementation-oriented decisions.

There is yet another source of information; the environment entities. These express constraints, norms and systems in the environment in which the task is performed.

If constraints and norms are specified, they can provide:

- qualifications of functionality requirements, setting absolute limits on acceptable behaviour, and
- qualifications of non-functional requirements, requirements that should not be violated.

Summary: The task model provides explicit and strong functional requirements on the application, mainly expressed within the task entities. Furthermore, there are several other model components that can express critical non- functional requirements

6.2.4 Agent Model

The aim of the agent model is to represent constraints and requirements on the agents performing tasks. Agents may be humans, performing tasks and entering into communication with systems. Agents may also be systems that exist or will be built. These agents perform tasks identified in the task model, and for agents to perform these tasks they must have some capabilities. These capabilities are necessary and sufficient requirements on agents performing the tasks.

The agent model contains entities called capabilities, where the requirements are recorded.

When agents are performing a task that will be automated, the capabilities give an indirect description of the requirements on this task. The form of these requirements can range from very simple characterisations to full-fledged specifications. The methodology itself does not enforce a certain level of description. The amount of information specified for capabilities will depend on the risks involved.

If a decision is made to build a system for a task that is not elaborated in the expertise model, and a decision is made to explicitly state the requirements on that task, this requirements statement can either be recorded in the description of the task in the task model, or in the capabilities entity for the agent performing the task.

When an human agent is involved in performing a task, but a decision has been made not to build an application for that task, the agent model will still provide indirect requirements on the design, as the agent-may have to communicate with the application, and the application must establish the right level of communication with the agent, given the capabilities of the agent.

Summary: the agent model can either provide full functional specifications (for agents to will become applications) or it can provides non-functional requirements on the design (for the case where the agent will remain as a human, and will participate in the task).

6.2.5 Communication Model

The communication model describes communication between different agents. When an application is to be designed for a task, it must have some interface with the external world, be it through sensors, terminals, networks, or inter-process calls.

For this reason the communication model will be of significant importance for the design model, specifically for the interaction subsystem of the application design.

There are two areas where the communication model provides explicit information that has a direct influence on the design:

- the representation of information being communicated
- the sequencing of elementary communication acts

In the ingredient entity of the task model, the semantical structure of communicated information was described. That semantical description is supplemented by a syntactical description in the information item entity of the communication model.

The information item provides two slots:

syntax: which describes the way in which the information structure (data structure) is to be presented by the sender to the receiver.

media: which describes on what media the syntax is represented.

When there is communication between two applications (two system agents), the media may be something like:

- a file
- an interprocess message
- a procedure call

The syntax could then be specified, respectively, in the form of :

- record structure definition
- message structure definition
- ordering and typing of the arguments in the call

If there is communication between a system agent and a human, then the medium must be something that is directly accessible by the human. This includes:

- graphical screens (output only)
- keyboards (input only)
- mouse (input only)
- printed paper (output only)

On different media like these, one should choose a (surface) syntax that maximizes the utility to the human. Hence the choice of media and choice of syntax are critical decisions that should take man-machine interaction aspects into account. The communication model provides a representation formalism where the result of these decisions can be recorded.

This kind of information, if available, will be very close to design decisions themselves, and consequently they are more or less directly usable in the design.

The other major area supported by the communication model is the way the simpler communication acts are composed into more complex communication patterns.

This is achieved by modelling flexible communication in terms of transactions, discourses, and transaction plans. The aim of these model components in the communication model is to provide a framework in which the elementary communication acts can be structured for optimized information exchange.

The parts of the communication model that describe or define the communication patterns (transaction, transaction plan, discourse) constitute a functional specification of the communication aspect of the application.

From the design point of view, this functional specification is very important. These specifications directly influence the design, but whether the structuring of the communication patterns represented in the communication model can be directly mapped to the design model depends on the choice of platform and architecture for interaction.

Summary: In the communication model, the information items provide detailed restrictions on design choices, while the transactions, discourses and transaction plans provide a functional specification.

6.2.6 Expertise Model

The expertise model is a description of the problem-solving behaviour for complex tasks. This behavior is described at different levels; the task knowledge level, the inference knowledge level, and the domain knowledge level.

As the whole purpose of the expertise model is to provide a detailed description of a complex task to be performed, the entire model definitely implies a functional requirement on a KBS component of the application.

Furthermore, as the model also provides a description of *how* the complex behaviour can be achieved, it implies possible ways of designing the KBS component.

Different ways of operationalising the KBS component have been identified. One approach that has been explored is called "structure preserving design", and is based on the paradigm that the structure of the expertise model should be reflected in the design of the application. This approach is described in Ch. 10.

As the expertise model relies on the context of problem-solving being defined in other models, it does not contain constituents that only imply non-functional requirements.

Summary: The expertise model provides detailed functional requirements on the application. There are no non-functional requirements implied by the model.

6.3 Summary of this chapter

We have in this chapter provided a road-map to the CommonKADS models, indicating where, in the total set of models, one can find information directly influencing design decisions. We also indicated whether such information provided or indirectly implied functional requirements, or whether this information only provided non-functional requirements.

Chapter 7

Design approaches

Olle Olsson, Ulf Sundin

In standard software engineering (SE) there are many different philosophies and religions concerning what is the best way to perform design work. In any book on SE, there is one or more chapter dedicated to presenting some approach to how design should be done. Unfortunately, it is not profitable to directly apply any of those approaches to the design problem in CommonKADS.

The reason for this is that CommonKADS is a model-based methodology, where everything is regarded as models. This means that we do not have standard requirements documents, or specification documents, documents that are organised to fit the software engineer more than any other professional role. The commonKADS models (except the design model) aim at providing excellent support for analysis of the problem and validation of our understanding of the problem.

The characteristic feature of CommonKADS design is that it takes it starting point in the models describing the problem. So how to make the jump from the models describing the problem to the model describing the design?

We do not have an approach to design that is enforced as *the* CommonKADS design approach. Such an approach must be well-founded in experience, and so far we do not have much experience in the use of the design model.

What we can do is to make a small map, identifying some paradigmatic approaches to design. We will not provide any complete elaboration of any of these. In the succeeding two chapters we describe two approaches and provide pointers to the literature.

We can identify a set of characteristics that is of use when trying to understand what options are available.

The first characteristic is whether a design library is used or not:

use library component: the design is created using components that are available in a design support library. Such components are located, selected, specialised, configured and glued into the evolving design

build from scratch: all components in the evolving design is created specifically for this design.

Another split can be seen in what direction design proceeds in the design model:

design top down: the application part of the design is constructed, then the architecture, then the platform is identified. This proceeds in a top-down fashion, from the parts closest to the user and the problem, down to the mechanisms supporting whatever is above them.

design bottom up: the platform and architecture part are evolved before the application design is initiated. This is from the bottom up.

We see the corresponding direction in the specification models:

specification top down: start your thinking about design by trying to understand how tasks in the task model might be turned into designs. Then proceed downwards, through tasks in the expertise model, inferences in the expertise model, and then the domain layer.

specification bottom up: start Your thinking about design at the domain layer, understanding what properties it has, what alternatives for design they admit, etc. Then continue up in the hierarchy.

An important distinction is the attitude to the structures that can be found in the specification models:

structure preservation: structures should be transformed into isomorphic structures in the design models.

structure elimination: structures providing control in the specification models should not be represented by any control structure in the design model

structure replacement: structures in the specification models provide information that is of importance, but these structures should not just be replicated in the design model. From the design point of view completely different structures might be more suitable.

These characterisations provide dimensions, allowing for 24 different ways in which one can characterise a specific design approach.

We will in the next two chapters have a look at two (compatible) alternative design approaches.

Chapter 8

Design by structure mapping

Olle Olsson, Ulf Sundin

In this chapter we will present an overview of how some specification model can directly indicate certain design routes.

The underlying insight is that some models are structured in a way that can bear directly on the structure of the resulting design. This partly depends on the fact that some decisions concerning the contents of the specification models are best seen as design decisions. That the specification models are based on design decisions has already been mentioned. What we will see below is that the results of some of these decisions are so strongly related to the structure of the design, that we should take these model structures as a starting point for parts of the design model.

An attractive feature of this approach is that the design will inherit parts of its structure from the structural decisions in the specification models. This approach is sometimes called *structure-preserving design*.

A design that inherits its structure from the specification model will exhibit some desirable properties, e.g.:

- the design will be easier to verify with respect to the specification model
- the design will be easier to maintain, as changes to the specification will be easier to map to the corresponding part of the design

In the sections that follow, we will illustrate how structures in the specification models can provide recommendations for structural decisions concerning the design.

As the specification models only describe aspects of the problem that is the focus of the CommonKADS development project, they contain information that can only be related to the *application design* in the design model[1]. When design work is restricted to application design, the guidelines below can be directly used. This will be true for projects where a standard platform and a standard architecture is enforced in the project as a restriction.

[1]This statement is not quite true. An example of the opposite is that the organisation model contains information about computing resources, and these will influence the decisions in the platform part of the design model

For project that will include design of new architectures or the use of specialised architecture, the guidelines may have to be revised, as the border between what is to be done in the application and what is to be done in the architecture is not a priori fixed.

8.1 Task Model

The structure in the task model that is easily related to structures in the design is the hierarchical decomposition of tasks into sub tasks.

The task decomposition is in effect a description of one way of performing a task in terms of performing a set of simpler tasks. This corresponds elegantly to standard structural software design techniques.

Tasks in the task model can be mapped to processing units (e.g., procedures), and the hierarchy of task decomposition can be mapped to the hierarchy of activations of such units. In the case of procedures, the task hierarchy will be a procedure-call hierarchy.

The control-structure slot of a task in the task model contains a specification of how the subtasks are coordinated. It will often be specified in procedural terms, and then the specification of the control structure can easily be transformed into a corresponding control structure for in the design model.

Due to restrictions in the architectures used, the design model has to be more restrictive compared to what is allowed by the task model. An example of this is where two subtasks of a task can be performed in parallel, which might be the case when these two subtasks are can be performed independently of each other. If the selected architecture for the design does not offer any way of specifying parallel and independent execution, some sequencing decision has to be made in the design model. When there is complete independence between two subtasks, the execution of their corresponding design components can be scheduled in any order.

For leaf tasks, we do not have control over any sub tasks. Leaf tasks only refer to primitive activities and their synchronisation. It is at this level that we must have complete information about functionality of the activities. For leaf tasks that are not further explored in the expertise model, we typically find very sketchy information. This sketchiness is mainly caused by elaboration effort being spent on the more intricate tasks and activities, especially on those explored within the expertise model.

If we, during design, find these examples of vaguely specified activities and tasks, their specification should be entered into the task model, to serve as the original repository of analysis information. I.e. detailed specification work is done in the task model on demand.

If the norms and constraints of the environment entities are specified in the form of sharp and exact constraints, then they can be used in either of two ways:

- either "design-in" procedural version of these constraints into the design of the task, to be evaluated at run-time, resulting in a generate-and-test behavior, or

- use these constraints as additions to the specifications, and perform an complete design that will provide the correct functionality. One can later use these constraints by apply a static analysis to the design to "prove" that the design will respect these constraints.

8.2 Communication Model

As described in an earlier chapter, the communication model can represent a structuring of the interaction between agents that has direct relevance for the design model.

The information item specification in the communication model can provide a detailed description of how data is to be represented in the communication act. This description provides direct indications for how the ingredients involved in the communication should be represented at the design level.

For technical reasons, the information specified in the **syntactic structure** and the *media* slots of the information item might not be completely valid for the current design. The degrees of freedom of specifying the syntactic and media aspects might be more restricted than was believed, due to the enforced use of some specific technology and/or system for realising the communication. This might cause some changes to the way the syntactic structure of ingredients will be designed in the design model.

The communication patterns are described in the communication model in terms of transaction plans, transactions, and discourses. These descriptions directly influence the corresponding structure in the design model.

A transaction plan describes the desirable orderings of a set of individual transactions. A transaction describes how a set of ingredients can be transferred from one agent to another. A discourse specifies a plan for how a specific transaction can be carried out as a set of single interactions.

These descriptions are in the communication model seen as descriptions on a cognitive level, and represents a conceptualisation of the structure of communication. The structuring of the communication model has been defined in this way, to make it possible to *represent* the conceptualisation of communication at a rather detailed but still implementation-independent level, and to make it possible to *evaluate* this conceptualisation from a cognitive point of view.

The task of design with respect to the communication model is to devise a design that will implement the specification expressed in the communication model. The detailed structure of the communication model will make this design task easier, but will not eliminate the need for design decisions. There are several reasons for this, and we will mention some here.

The notation for transaction plans does not uniquely define the way in which in which transactions will be synchronized. The notation specifies *minimal requirements* on transaction orderings, and should therefore be seen as constraints restricting the set of possible orderings.

From a design point of view, it might be necessary or suitable to restrict the set of possible transaction orderings further. This can be done by specifying the *preferred* ordering, which should of course be an ordering that respects the given constraints. What this amounts to is really a partial evaluation of the transaction plan, in the context of other design decisions. E.g. some other design decision may have sequentialised the execution of some tasks that, from an analysis point of view, could have been performed in any order, or in parallel. Knowing that these tasks now have a fixed ordering may make it possible to specialise the transaction plan to a fixed ordering of transactions.

There is another way of specifying transaction ordering restrictions, by specifying a value for the **constraints** slot of a transaction. As stated in the Communication Model document, the constraints slot contains a "distributed version of the transaction plan, defining the prerequisites that must be fulfilled for a transaction to be performed". As the transaction plan and the constraints slot of the involved transactions should express the same kind of information, it seems to be a matter of taste whether one specifies the constraints only in the transaction plan, or one specifies them only in the constraints slot of the individual transaction types. From a design point of view, a preference can be expressed; if the communication architecture supports the representation of transaction plans, these plans should be represented in the corresponding plan objects in the design model, with a structure that clearly reflects the plan as specified in the communication model. If the architecture does not provide for representation of plans, then the distributed version (use of the constraints slot) is to be preferred, as there is a design representation of transactions.

A transaction represents a package of ingredients that needs to be communicated between two agents. The structure involved here, from the design point of view, is mainly the 'involves' relation between the transaction and its ingredients. This has a natural correspondence to the design model, in terms of the transaction design object involving a set of ingredient design objects.

Discourses as specified in the communication model can express ways of ordering the communication of individual ingredients involved in a transaction. From this structure it is easy to identify the set of all possible sequences of atomic ingredient transferral acts. As the design should permit all these sequences to appear, the structure of the discourse should be mapped to the corresponding discourse structure in the design model.

In contrast with the transaction plans, the notation for discourse does not require an advanced planner to plan for what the actual sequence will be in the discourse. The "plan" expressible in the discourse notation can be represented in a finite directed graph, where branches in the graph corresponds to simple decision points. A simple finite state machine is able to sequentialise the individual ingredient transferral acts in a transaction, based on the discourse description. For this reason, the structure of the discourse should be isomorphically represented in the design model.

For technical reasons, a different grouping of ingredients might have been preferred. There may be limitations on the actual media used for interaction, limitations that restricts the number of ingredients that can be represented at one and the same time. As the concept of transaction represents a conceptual need for communication, caused by tasks requiring information from some agent, or tasks having information for some agent, the extent of the transaction in terms of ingredients should not be changed.

Instead, the discourse structure can be used to rearrange communication to satisfy the technical restrictions. By inspecting the information items describing the ingredients, and by simulating the discourse structure, one can identify how the discourse can be rearranged to partition the ingredient transferral acts, so that the technical limitation will be satisfied.

8.3 Expertise Model

Design of the application corresponding to an expertise model can be complex, due to the multidimensionality of the expertise model structure itself. As the goal of the expertise model is to provide a framework for understanding complex problem-solving tasks, describing the competence and be explicit about problem-solving constraints, it is not at all obvious that it is a suitable staring point for a design.

Experience in building KADS-based applications have shown that it may not be so difficult after all. The structuring of the expertise model into the three layers (domain layer, inference layer, task layer) gives a first partitioning of the expertise model, and this can be mapped to corresponding partitionings of the application design.

The inference structures in the inference layer can be interpreted as i/o dependencies between simpler processing units, each of which can be designed as a small procedure.

The task structures in the task layer includes task decompositions analogous to the task decomposition in the task model, and can be implemented in similar ways. The task structure provides the control aspect of the application, activating appropriate inference procedures when suitable.

The domain layer is a bit trickier to design, as it contains a large number of small items of knowledge, and there is no major structure in it that is obviously useful as a basis for a design structure. Furthermore, the domain layer cannot be completely designed without understanding how it is related to the inferences (used by the inferences). There are, however, guidelines for how to choose a design of the domain layer and relating it to the inference layer (see [Terpstra, 1993], [Schreiber, 1992]).

As the expertise model of CommonKADS is unique to this methodology, there is a need for advice on how to produce a design given an expertise model. This topic has been documented in a number of publications (e.g., [Schreiber, 1992], [Duursma *et al.*, 1992]). The guiding principle in those works is "structure-preserving design" of an expertise model. A following chapter, Structure-Preserving Design of the Problem Solver, discusses in more detail aspects on this approach to design.

8.4 Structure Mismatch Between Specification Model and Design Model

In the sections above, we indicated how the structures found in the specification models can be transformed into structures in the design model. Such structural isomorphism brings several desirable consequences, as stated in the beginning of this chapter.

A problem that has only been touched upon is the problem where the structure in the specification models cannot be isomorphically mapped to corresponding design structures. We have given some examples of this problem, and indicated in what ways a suitable design structure can be achieved.

An effect of choosing a design structure that differed from the structure in the specification models is that we loose some of the desirable properties of having a structure-preserving design. There are two ways of handling this situation:

- accept that the two structures are not isomorphic, but be careful to document that they *are* different, and provide the rationale for the deviation.
- try to reestablish an isomorphism by adapting the structure in the specification model to the structure in the design model. The problem may be caused by incomplete information being available at the time the specification model was built. During design, more knowledge may be available, and hence a revision can be made of the "design" choice in the specification model. An example of this is that when

the discourse structure of a transaction was defined, the limitations of the media to be used were not completely known. Had correct knowledge been available, a different discourse structure would have been defined.

The second alternative is yet another illustration of the fact that the specification models represent design decisions, and that such decisions may have to be revised, when some relevant piece of knowledge becomes available.

I.e. revising the specification models so that there is a clear correspondence to the design model may not at all be a retroactive adjustment of the specification to what was actually designed. It is more a matter of making the specification model better reflect our current understanding of the problem and its context.

8.5 Summary of this chapter

In this chapter we have presented the idea of a design approach that reuses as much as possible of the structure that is present in the specification models. We have also illustrated in some detail how the task model and the communication model contains structures that can be used as a starting point for structuring in the design model. We have also pointed to the expertise model as a good candidate for structure-preserving design, but has not elaborated on this, as it will be presented in a separate chapter.

Chapter 9

Design by flattening

Olle Olsson, Ulf Sundin

In the previous chapter, we saw how the structure in the specification models could be transformed into corresponding structures in the design model. That approach to design exhibits a number of desirable consequences, as has been indicated.

In this chapter we will look at an approach that does the opposite; eliminating as much structure as possible, as this can in some ways simplify design and implementation.

We will only sketch this approach for the case of the expertise model, but e.g. the task model can be handled in analogous ways.

The expertise model contains the three layer task-layer, inference-layer, and domain-layer

The task and inference layers are the ones that most closely reflects the major problem solving behavior, as they can informally be interpreted in a procedural way. We should be aware of the fact that this is not the only way to understand the task and inference layers. But from a design point of view it can be useful to have this operational perspective on these parts of the expertise model. The task layer provides the control aspect and the individual inferences can be seen as black-box computational units. During problem-solving the task layer activates the appropriate inference, the inference performs its computation, and the task selects the next inference to activate. As the task layer has a hierarchical structure of tasks and subtasks, we can also talk about tasks activating other tasks.

A task "contains" knowledge about what inferences it is responsible for, and when to activate which of its inferences. The task itself does not do any serious computation. it is mainly concerned with ordering the inference activations. In its task body the control structure represents this knowledge. We now assume that this control structure is expressed in a procedural form, analogous to the the types of control structures present in standard procedural languages.

The control structure, being of this simple form, and typically having a very simple syntactical structure, can be rephrased in different ways without changing the ways in which the inferences are activated. We will not go into technical details[1], but just state

[1]see: C Boehm, G. Jacopini: Flow diagrams, Turing machines, and languages with only two formation rules (Communications of the ACM, vol 9, no 5, pp 366-371, 1966)

that by suitable introduction of new variables representing states, we can define a simple procedure (or algorithm) that activates inferences in the same way as the original procedure, and where for each inference there is an explicit condition stating when it can be activated. Having arrived at this form, it is now a simple step to move these conditions into the inferences, making them responsible for checking if it is appropriate for them to activate themselves.

In this way we can eliminate the task that controls inferences. And in exactly the same way, we can eliminate the entire hierarchy of tasks, resulting in a *flattening* of the control structure of the expertise model. The entire task hierarchy collapses into the inference layer, each inference receiving the control information that concerns itself.

Expressed in other words; instead of having centralised all control in the task layer, now we have this control knowledge distributed among the inferences.

Now that we do not have any structure left in the expertise model (ignoring the domain layer for the moment) we can, from a design point of view, see each inference with its own activation information, as small processing units, mapping directly to small design objects.

What we have done is to perform a transformation of the expertise model into another "expertise model", where these are equivalent from a behavioral point of view. The flat version of the expertise model can, as just said, easily be mapped to the design model.

So what is the use of this flattening? There is one interesting use, and that is in the context of design architectures supporting agents operating "independently" of each other. An example is a blackboard system, where the knowledge sources operate without being directly synchronised by some other agent. So if we want to use a blackboard architecture, we must have some technique for transforming the expertise model into a design that can optimally use the strength of the backboard, and flattening can be such a technique.

There is yet another use of flattening. A syntactical approach to flattening the control structures will preserve the sequencing of inference actions. This is caused by the fact that if the task control is expressed in standard procedural control primitives, then we have a single locus of action in the task, corresponding to the program counter for a single-processor. Syntactic flattening will preserve this characteristic. This might be useful from a requirements point of view, if e.g. explanations based on traces are to be presented to a user. The single locus of action guarantees that the trace of interest is the sequence of events that just happened.

But we might profit from having a bit more freedom in the control. Some orderings of inferences might be enforced, e.g. due to input/output dependencies. Other orderings are not of that necessary kind. If we can, in the flat version, eliminate all unnecessary synchronisations and only keep the necessary ones, then we have the option of performing inference chains in parallel. This might be an interesting option for certain kinds of applications, and certain kinds of platforms/architectures, even opening up the possibility to use multiprocessor systems for parallel search in large search spaces. The utility of parallel machines for these kinds of applications has been proved.

One last remark. We saw in the preceding chapter how structures can be mapped to isomorphic structures in the design model. In this chapter we have seen how structures can be flattened. Combining these two approaches we get two different cases shown in Figure 9.1.

As see, we can apply the flattening technique on the expertise model and then do a (simple) mapping, or do a structure-preserving mapping from the expertise model to the

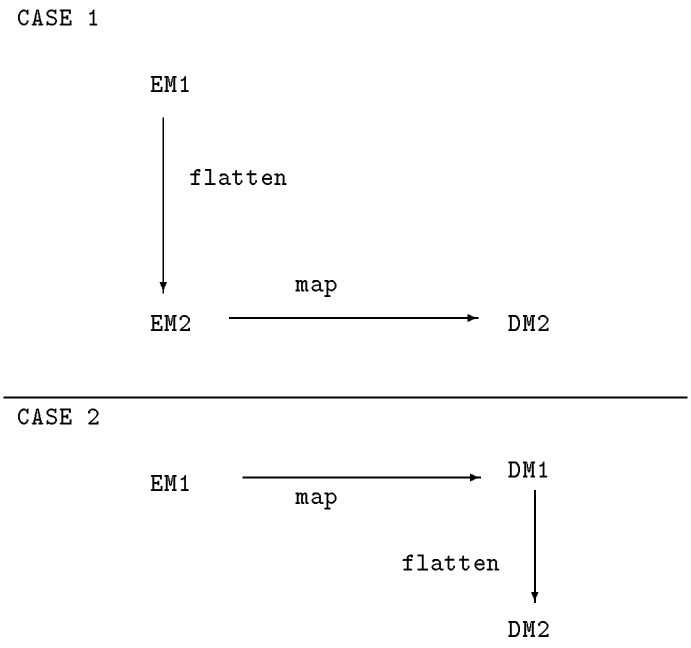


Figure 9.1: Combining mapping and flattening

design model, and then do the flattening in the design model.

Chapter 10

Structure-Preserving Design of the Problem Solver

Guus Schreiber

This chapter provides a presentation of how the concept of structure-preserving design can be applied to the design of the problem solver component in an application. The major concern for the task of designing the problem solver is how to operationalise the expertise model, and this is the perspective taken in this chapter. In some circumstances other models may influence certain problem-solver design decisions, e.g. the need for explanations as expressed in the communication model. Such issues will be touched upon in this chapter.

10.1 Rationale for Structure-Preserving Design

Design concerns two important tasks (i) the specification of an system architecture and (ii) the selection of appropriate representations and computational techniques (the definition of the virtual machine(s)).

In principle, the designer is free to make any set of design decisions that results in meeting the requirements formulated during analysis of the problem solving requirements, as laid down in the expertise model. However, from a methodological point a *structure-preserving design* should be strongly favoured. With "structure-preserving" we mean that the *information content and structure* present in the knowledge-level model is preserved in the final artifact. For example, it should be possible to retrieve from the final system both the domain-knowledge structures specified during analysis as well as their relations with inferences and/or knowledge roles. Design thus should be a process of *adding* implementation detail to a knowledge-level model. The knowledge-level model is in fact interpreted as a skeletal architecture for the system.

Thus, preservation of information is the key notion. For this purpose, we investigate in the next section in more detail the nature of the information in the expertise model. In Sec. 10.1.2 we discuss the main advantages of the structure-preserving approach.

In Sec. 10.2 we take the "executable specification" view on the expertise model and try to define a skeletal architecture for supporting structure-preserving design of the problem solver. In Sec. 10.3 we investigate how the structure-preserving approach influences the computational decisions.

10.1.1 Types of information in the expertise model

In this section we take a second look at the structure of the expertise model and characterise what kind of information it contains with respect to the three perspectives data, function, and control:

Functional perspective

The functional perspective (the *how* viewpoint) on an application is about what functional transformations are performed by the application, typically described by a hierarchy of functions.

In the expertise model this perspective is represented both in the inference knowledge and in the task knowledge. Inferences constitute the leaves of the functional decomposition tree. Tasks represent the higher-level functions.

Data perspective

The data perspective (the *what* viewpoint) on an application is about what data the application operates on, i.e. the data the that are transformed by the functions of the application.

The domain knowledge represents a specification of all domain-specific data and forms the major part of the data perspective.

The dynamic and static knowledge roles in the inference knowledge are also part of the data view, but these data are specified in an indirect way: i.e. by mapping functional names onto domain-knowledge elements. One could say that the name of a knowledge role specifies a functional object (e.g. *hypothesis*) and that the *mapping* onto domain terms (e.g. *hypothesis* → domain class *disease*) assigns a data structure to such a functional object. The mapping provides in fact the link between the functional perspective and the data perspective. The situation in COMMONKADS is thus more complicated than in conventional data-flow diagrams, where the names of the data flows refer directly to elements of the data model.

The control terms (part of the task knowledge) specify similar functional objects, but through a multi-step mapping (e.g. the control term *differential* maps onto a set of objects of type *hypothesis*, which in turn maps onto a class of domain objects.)

Control perspective

The control perspective (the *when* viewpoint) on an application is about the principles determining when what functions of the application are to be performed.

The control perspective is present in the task knowledge (the control structure description of a task) and in the strategic knowledge.

By relating a description of control directly to a task, an explicit link is made between the functional perspective and the control perspective. This is not by definition true for the control specified in strategic knowledge, as COMMONKADS provides no strict guidelines for the structure of this type of knowledge,

Table 10.1 summarises the information contained in the various parts of the expertise model with respect to three viewpoints.

Category	Element	Perspective
domain knowledge	concepts, properties relations, structures, etc.	data data
inference knowledge	inference dynamic knowledge role static knowledge role	functional functional → data functional → data
task knowledge	control terms sub-tasks control structure	functional → data functional control → functional
strategic knowledge		control

Table 10.1: Characterisation of the information contained in a expertise model with respect to the three perspectives on systems: data, function and control.

The expertise model is thus not biased towards one particular perspective, although part of the functional view, i.e. the inference structure is often the starting point for building the model. The resulting architecture is thus a mixture of different types of components, each emphasising different perspectives of the system.

10.1.2 Advantages of structure-preserving design

There are a number of reasons for following a structure-preserving approach to design:

Reusability of code
Structure-preserving design prepares the route for reusability of code fragments of a KBS, because, in Smith's terms the "semantical attribution" [Smith, 1985] of code fragments is explicit. Reusable code fragments can be of various types and grain size, ranging from implementations of inferences to implementations of an aggregation of inferences plus control knowledge. The modular structure of COMMONKADS expertise models facilitates this type of reusability.

Maintenance
The preservation of the structure of the knowledge-level model makes it possible to trace an omission or inconsistency in the implemented artifact back to a particular part of the model. This considerably simplifies the maintenance of the final system. It also facilitates future functionality extensions. Experience with the Fraudwatch system (cf. [Killin *et al.*, 1991]) indicates that systems built in this fashion are much easier to maintain than conventional systems.

Explanation
A structure-preserving approach facilitates the development of explanation facilities

that explain the reasoning process in the vocabulary of the knowledge-level model. For example, for some piece of domain knowledge it is possible to ask:

- in which elementary problem solving steps it is used and which role it plays in this inference;
- when and why it is used to solve a particular problem (control knowledge).

As the knowledge-level model is phrased in a vocabulary understandable for a human observer, a structure-preserving design can provide the building blocks for "sensible" explanations. This feature has been demonstrated by [Clancey and Letsinger, 1984] in the NEOMYCIN system. Several researchers have proposed generic strategies for generating such explanations from knowledge-level descriptions [Neches et al., 1985, David and Krivine, 1990, Sprenger, 1991]. Generic explanation tools supporting questions like *why/when/how inference* by exploiting the structure-preserving property of the code.

In EES [Neches et al., 1985] the model information is preserved in a separate "development history" generated by a program writer. This database stores facts about the relation between model elements and code fragments (i.e. OPS5 rules) generated from these model elements. This "compiler" approach to information preservation is attractive from an efficiency point of view.

Knowledge acquisition support

Given a structure-preserving design, the knowledge-level description can fulfill the role of semantic information about pieces of code of the artifact. This additional information can be used to support knowledge acquisition in various ways. Some examples:

- One can construct editors for entering domain knowledge directly into the system which interact with the user in the vocabulary of the model, similar to systems like MOLE [Eshelman, 1988b].
- One can build debugging and refinement tools which spot errors and/or gaps in particular parts of a domain knowledge base by examining its intended usage during problem solving.
- It is possible to focus the use of machine learning techniques to generate a *particular type* of knowledge, e.g. abstraction and specification knowledge (cf. the Acknowledge project [Shadbolt and Wielinga, 1990, van Heijst et al., 1992]).

An example might help to illustrate the advantages of a structure-preserving design. Suppose we have in a medical application some pieces of domain knowledge of the following form (here phrased as logical implications):

temperature $\geq$ 38.0 $\rightarrow$ fever = present
diastolic-pressure $\geq$ 95 $\rightarrow$ blood-pressure = elevated
blood-pressure = elevated $\rightarrow$ hypertension = present

Suppose also that we have the following two inferences in our expertise model applying such pieces of domain knowledge in the reasoning process:

(i) An *abstraction* inference in which a finding, e.g a quantitative property like the temperature of a patient, is abstracted into a more abstract finding such as fever.

(ii) A *specify* step which defines an inference that can be used to find-out which observable should be asked to the user once it becomes known that the patient has, for example, a fever.

Although the validity of the model is not the issue here, it might be useful to add that physicians usually want to get an answer to the question "What is the temperature" when they learn (e.g during physical examination) that a patient has a fever, as the precise value can be important in other parts of the reasoning process.

Fig. 10.1 depicts a simple inference structure containing these two inference steps.[1]. The italic annotations of the dynamic knowledge roles denote the domain-knowledge elements that could fulfill the role of finding or observable. The domain implications listed above are used by both inferences as static knowledge roles. The inference structure also contains an additional step (the oval with the dashed border), which denotes a transfer task for obtaining the value of an observable (e.g. "What is the value of the temperature").

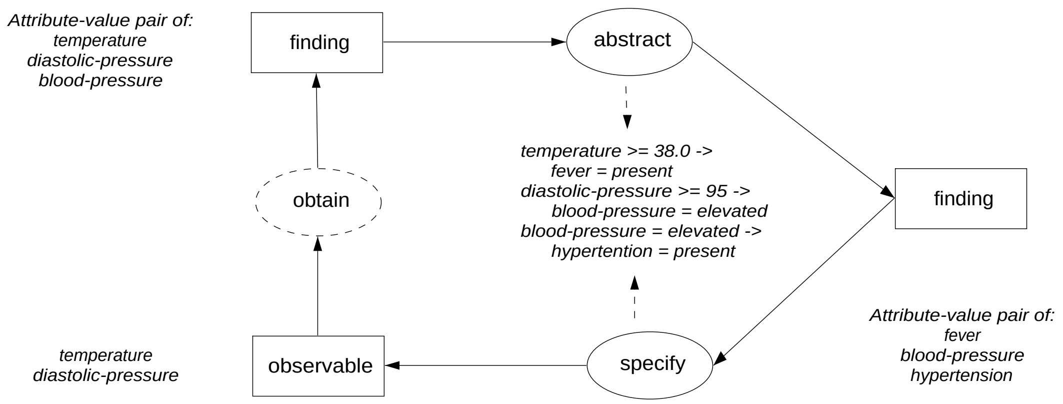


Figure 10.1: Two inferences in a medical application with some examples of related domain knowledge. Both *abstract* and *specify* use the same domain knowledge, but for different purposes. The dashed oval denotes a transfer task.

A structure-preserving design dictates that this two-fold use of essentially the same pieces of domain knowledge is made explicit in the final implementation. For example, implementing these domain expressions in duplo as abstraction and as specification rules would violate the structure-preserving property. It would give rise to knowledge redundancy (the same knowledge is present twice in the system) and can lead to serious maintenance problems (when specification knowledge is changed, the abstraction knowledge needs to be changed as well). For a knowledge acquisition tool it could mean that the user needs to enter the same piece of knowledge twice (or it would require an undesirable *ad*

[1] This inference structure is in fact a small fragment of the KADS inference structure for heuristic classification (see [Schreiber, 1992a]).

hoc adjustment of the tool). For explanation purposes it is important to be able to explain the different usage of these implications. At different points in the reasoning process the explanations about the role of these implications can vary.

Structure-preserving design is currently also being advocated in conventional software engineering, especially in the area of object-oriented modelling and design [Rumbaugh et al., 1991, Coad and Yourdon, 1991]. The motivation there mainly concerns reusability and maintenance.

10.2 Skeletal Architecture

In structure-preserving design, the idea is to use the expertise model as a skeletal architecture of the artifact. In fact, most knowledge engineers using COMMONKADS think about the expertise model as some (semi-)executable specification. One could say that they have internalised a particular architectural interpretation of the model and use this to explain how the model they have built can solve a particular problem. However, the information in the expertise model is incomplete from the executable-specification point of view. This is not surprising, as this model is primarily meant for analysis purposes and not for design. The main ingredients that are missing and that need to be considered in architectural design are *inference methods*, *domain indexing and access functions*, *working memory*, and *human-computer interface functions*:

Inference method

Inferences specify the nature of its input and output (the dynamic knowledge roles) and the domain knowledge used in deriving the output from the input (the static knowledge role). Inferences do *not* specify *how* the inference will be achieved. This *how* description is typically something that has to be added during design. During analysis, the knowledge engineer often takes, what one could call, an *automated-deduction view* on a particular inference: the knowledge engineer specifies a inference in such a way that she knows that it is possible to derive a conclusion, given the available knowledge, no matter how complex such a derivation in practice might be. In analysis, the emphasis lies on a competence-oriented description: can I make this inference in principle, and what is the information I need for making it happen. An inference method specifies a computational technique, that actually does the job.[2] Some example inference methods mentioned in [Breuker *et al.*, 1987, p. 41] are inheritance, empirical association, matching algorithms, and generalisation.

One can take the view that inference methods are part of knowledge sources and thus should not have the status of separate architectural components. However, the relation between inferences and inference methods is not one-to-one. Several inferences may apply the same inference method, but for different purposes. For

[2] If the expertise model is fully specified in some strictly formal language such as ML^2 (cf. [van Harmelen & Balder, 1992]) it is in principle possible to use a dedicated theorem-prover for executing the model. In practice however, some inferences involve computations that are, given the state of the art, computationally intractable within such an approach . A theorem-prover, $Si(ML)^2$ [Aben, 1993], has been developed for simulating the problem-solving behaviour of a particular model, providing pragmatic short-cuts for intractable parts. This theorem-prover is meant to be used for model validation purposes and is not expected to fulfill the role of final system.

example, in the StatCons system for statistical consultancy two inference methods realised in total eight inferences [de Greef *et al.*, 1987, pp. 73-99]. Also, in the Sisyphus application described in [Schreiber, 1992] one method is used to realise three different inferences.

The reverse can also be true, namely that one inference function is realised through multiple methods. Thus, incorporating inference methods into operational inferences prevents making full use of the reusability concept.

Domain access and indexing

One of the subtle points of the expertise model framework is the use of functional names (i.e. dynamic and static knowledge roles) to describe the inference process. In the inference specifications the mapping between functional names and domain-specific terms is indicated. During design, one has to construct for these inference/domain mappings *indexing* mechanisms for the (operational) domain knowledge-base. Domain-access functions use this indexing information to retrieve the required domain knowledge for carrying out a particular inference. These access functions ensure that system elements that realise inferences (the "inference functions", see further) can be specified fully in a manner that is independent of the application-domain. This is a key point with respect to reusability.

Working memory

During analysis one is (and rightly so) sloppy in defining storage of the run-time data. This is mainly because a full specification of this type of information requires a detailed description of all kinds of data manipulations. The working memory (with which we mean the database of the run-time results of inferences and tasks) is only specified implicitly, in particular through dynamic knowledge roles and control terms. During design, one has to define however explicitly the nature of working memory, possibly adding other types of run-time information as well: e.g. which tasks have been executed etc.

Human-computer interface functions

Although the emphasis lies in this chapter on the impact of the expertise model on the design, a few remarks are in place here about the components that are concerned with the interaction of the system with external agents. The communication model contains the detailed specification of this part of the system. In the skeletal architecture (cf. Fig. 10.2) such components appear as human-computer interface (HCI) functions.[3] These functions implement transfer tasks such as obtaining a value and presenting a solution.

It should be noted that the communication model can interact with the specification of control in the expertise model. For example, it can be the case that the communication model specifies a desired interaction strategy in which obtaining the value of certain observables should involve asking also for values of related observables (e.g. because this increases the plausibility of the line of questioning of the system). In fact, the communication model often gives rise to an adaptation of the

[3] The skeletal architecture makes the simplifying assumption that the external agents the system will deal with are all humans.

control knowledge as specified in the model. Typically, it influences the design of task knowledge and strategic knowledge specified.

Fig. 10.2 depicts a typical skeletal architecture for structure-preserving design. We briefly discuss its various components below, except for the HCI functions which fall outside the scope of this chapter.

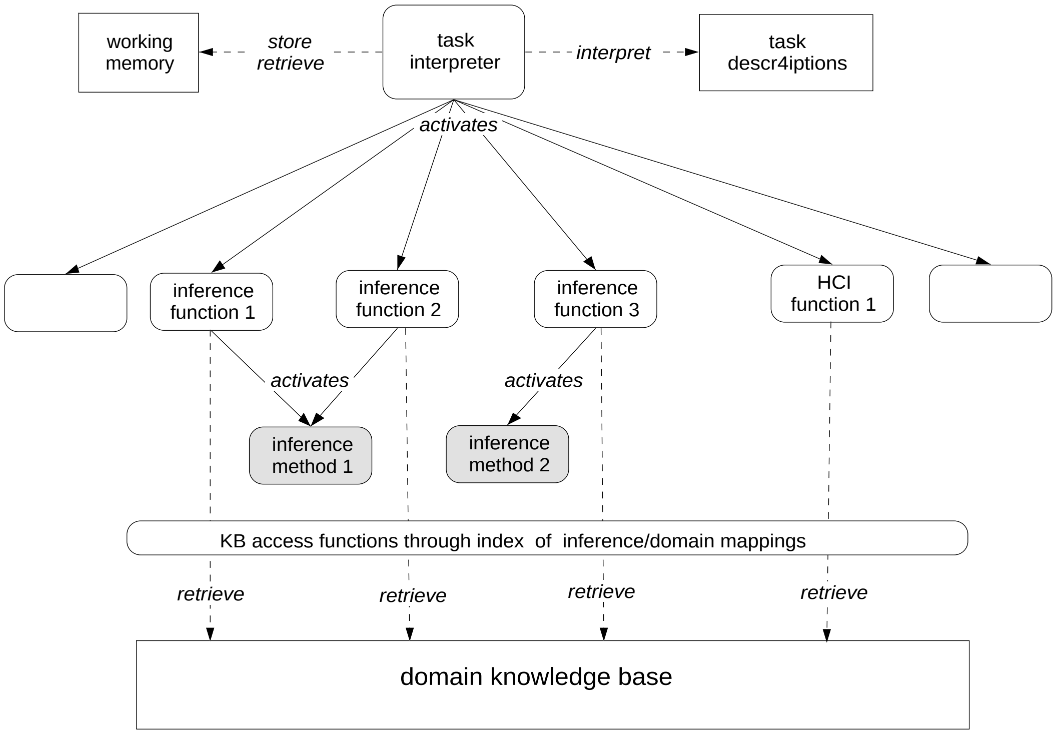


Figure 10.2: Skeletal architecture supporting structure-preserving design. The boxes with rounded corners denote system components that are active during reasoning; the sharp-edged boxes denote data stores. Non-dashed lines describe a control operation (*activates*) between components; dashed lines a data operation (*store*, *retrieve*, *interpret*).

Domain knowledge-base and access functions The domain knowledge-base contains a declarative symbolic representation of the domain-specific knowledge. It contains both the actual domain expressions (e.g. relation tuples, concept instances) as well a description of the structure of the domain expressions (the operationalised domain schema).

This schema is used by the knowledge-base access functions to retrieve certain types of domain knowledge, using the mapping specifications provided by dynamic and static knowledge roles. The access functions should be able to handle requests such as 'retrieve all abstraction knowledge" or "retrieve a domain entity that can play the role of observable".

The operationalised domain schema can also be used for explanation purposes (e.g. providing an answer to the question: "what domain knowledge was used in making this inference") and for the development of domain-knowledge editors that interact with the user in a domain-specific vocabulary, similar to OPAL [Musen *et al.*, 1987].

Inference functions and inference methods We use the term *inference function* to denote the design counterpart of inferences. Inference functions should contain the same information as described for inferences; types of input, output, and nature of the domain knowledge used (all in domain-independent terminology). In addition, every inference function defines how a particular *inference method* can be activated to realise the inference and what type of information needs to be retrieved (using the domain access functions) for successful execution of the inference method. In Sec. 10.3, we provide an example operationalisation of the inference functions for the abstract/specify example. More elaborate examples can be found in the description of the Sisyphus application [Schreiber, 1992].

Task interpreter, task descriptions, and working memory The task interpreter is the central control unit of the system in this architecture. It uses a declarative description of control structures and stores/retrieves run-time results of the reasoning process in/from working memory. The complexity of the task interpreter can vary, depending on the technique being chosen for implementing this component. In the simplest case, it is a straight-forward interpreter for COMMONKADS-type control structures. A more complex technique would be a blackboard-type of control technique. Choosing a particular control technique is discussed in more detail in Sec. 10.3.

10.2.1 Meta-level vs. object-level inferencing

The skeletal architecture shown in Fig. 10.2 has a meta-level flavour, in which the domain knowledge base represents the object-level and the inference- and task-elements represent the meta-level. Van Harmelen distinguishes three types of meta-level architectures based on the *locus of action* [van Harmelen, 1989]:

- Pure meta-level inferencing: all computation is carried out at the meta-level. The object-level has no interpreter and is only accessed to retrieve information.
- Pure object-level inferencing: the meta-level has no interpreter of its own, but is active at fixed points during the computation through the execution of predefined meta-predicates.
- Mixed inferencing: both object-level and meta-level have an interpreter. The object-level interpretation is a black box for the meta-level.

Within this classification, the skeletal architecture of Fig. 10.2 can be characterised as either pure meta-level or mixed, depending on whether the retrieval of domain knowledge involves computations in the domain knowledge-base. In the Sisyphus application the architecture is a mixed one: the domain knowledge-base contains additional axioms which are interpreted by the domain access functions: e.g. knowledge about properties of relation types such as transitivity is used to infer tuples of a relation.

10.3 Computational Decisions

In this subsection we discuss some more detailed design decisions in a structure-preserving approach. The scope of the section is limited to decisions with respect to elements of the expertise model. A more detailed description of these decisions can be found in [Schreiber et al., 1987, Schreiber et al., 1989].

10.3.1 Inference and domain knowledge

For each inference a corresponding computational technique needs to be selected that can realise this inference. A technique consists of three types of elements: (i) an algorithm, (ii) input-output data structures and possibly additional temporary data structures, and (iii) a representation of domain knowledge. The algorithm embodies the method for realising the inference and specifies the local, symbol-level control [Schreiber *et al.*, 1991a].

As remarked before, a number of groups of computational techniques have been developed in AI research, such as production systems, state-space search, parsing, classification and matching. These groups can be viewed as computational paradigms. Detailed studies have been performed to unravel the criteria for choosing a technique within one group such as hierarchical classification [Goel *et al.*, 1987] or automated deduction [Reichgelt & van Harmelen, 1986]. In [Schreiber et al., 1989] criteria for chosing a particular type of technique are discussed. Often, knowledge engineers use within one system only one or two types of techniques. For example, in NEOMYCIN [Clancey, 1985] four production system techniques are provided. Each inference[4] applies one of these production system techniques. Applying only one type of technique, such as production systems, in one particular KBS minimises the interaction problems with the design of other parts of the system (in particular the domain knowledge-base, as it requires just a single representational formalism), but apart from that there is no compelling reason to adhere strictly to this approach.

A crucial design decision concerns the choice of the representation technique(s) for the domain knowledge. Often, the nature of the knowledge described in the domain schema indicates suitable symbolic representations. If the knowledge engineer works within a particular paradigm, such as production systems, the choice of the representational technique is usually obvious: the domain representation technique is the same as the representation used by the chosen type of computational technique.

With respect to the structure-preserving principle, the most difficult part is to preserve the information about the functional names (knowledge roles), the domain specific names and their mappings. As the analysis in the previous section has shown, these inference-to-domain connections specify in fact the relation between the functional perspective and the data perspective on the system and constitute crucial areas for reusability etc.

To illustrate how structure-preserving design can be achieved in this respect, we present an example of a simple Prolog implementation of the inferences in Fig. 10.1. A full listing of the code, together with some sample traces, can be found in Appendix B.

Example fragment of an expertise model We assume that the inferences in Fig. 10.1 are specified in the following way in the expertise model:

inference *abstract*;
  input-roles:
    finding → "some expression about patient-data" ;
  output-roles:
    finding → "some expression about qualitative-data" ;
  static-roles:

[4] NEOMYCIN distinguishes only one "inference" type: *apply-rule*. The lowest level of meta rules fulfills however a similar role as inferences in COMMONKADS, namely performing the computations using domain knowledge.

```
    abstraction-knowledge →
      "< relation(qualitative-abstraction), relation(definition) >" ;

inference specify;
  input-roles:
    finding → "some expression about qualitative-data" ;
  output-roles:
    observable → "some property of qualitative-data" ;
  static-roles:
    specification-knowledge →
      "< relation(qualitative-abstraction), relation(definition) >" ;
```

These inferences use the same domain knowledge (see the static role), but for different purposes.

The domain knowledge is specified in the expertise model using the conceptual modelling language (CML) defined in [Wielinga et al., 1993]:

```
concept patient-data ;

concept quantitative-data ;
  sub-type-of: patient-data ;
  properties:
    temperature: number-range(35, 42) ;
    diastolic-pressure: number-range(0, 300) ;

concept qualitative-data ;
  sub-type-of: patient-data ;
  properties:
    fever: {absent, present} ;
    blood-pressure: {normal, elevated} ;
    hypertension: {absent, present} ;

relation qualitative-abstraction
  argument-1: expression(quantitative-data) ;
  argument-2: expression(qualitative-data) ;

relation definition
  argument-1: expression(qualitative-data) ;
  argument-2: expression(qualitative-data) ;

domain-model abstraction
  parts: abstraction-relation :
    tuple(qualitative-abstraction);
  axioms:
    qualitative-abstraction(temperature ≥ 38.0, fever = present)
    qualitative-abstraction(diastolic-pressure ≥ 95, blood-pressure = elevated ) ;

domain-model definition
  parts: definition-relation :
    tuple(definition);
  axioms:
    definition(blood-pressure = elevated, hypertension = present) ;
    qualitative-abstraction(diastolic-pressure ≥ 95, blood-pressure = elevated ) ;
```

The DDL description specifies two types of attributes of a patient and two relations that express relations between expressions about such attributes.

Representation of domain knowledge For this example we use a set of Prolog predicates that allow an almost direct translation from CML statements onto the chosen knowledge representation. This representation was also used for the implementation of the Sisyphus application [Schreiber, 1992]. The main idea is to keep the domain knowledge as much as possible in the form of a declarative theory of the domain, without any particular commitment towards specific use during reasoning.

```
% concept(Concept name, Supertypes)

concept(patient_data,       []).
concept(quantitative_data,  [patient_data]).
concept(qualitative_data,   [patient_data]).

% property(Concept,         Property name,       Valueset)

property(quantitative_data, temperature,         numberrange(35.0, 42.0)).
property(quantitative_data, diastolic_pressure,  numberrange(0, 300)).
property(qualitative_data,  fever,               [present, absent]).
property(qualitative_data,  blood-pressure,      [normal, elevated]).
property(qualitative_data,  hypertension,        [present, absent]).

% relation(Relation name,   Type first argument,     Type second argument)

relation(qual_abstraction,  expr(quantitative_data), expr(qualitative_data)).
relation(definition,        expr(qualitative_data),  expr(qualitative_data)).

% tuple(Relation name,   [First argument,           Second argument])

tuple(qual_abstraction,  [temperature >= 38.0,       fever = present]).
tuple(qual_abstraction,  [diastolic_pressure >= 95,  blood-pressure = elevated]).
tuple(definition,        [blood-pressure = elevated, hypertension = present]).
```

Automatic generation of this representation from the CML would require little effort.

Domain index The domain index specifies the mappings from inference-level names (see next paragraph) onto domain-specific data types in the knowledge-base.

```
domain_index(expression, finding,       [ expr(patient_data) ]).
domain_index(entity,     observable,    [ property(patient_data) ]).
domain_index(relation,   abstraction,   [ relation(qual_abstraction)
                                        , relation(definition) ]).
domain_index(relation,   specification, [ relation(qual_abstraction)
                                        , relation(definition) ]).
```

The predicate *domain-index* has three arguments, namely:.

1. The inference-level data type (one of *entity*, *relation*, and *expression*).
2. The inference-level name (e.g. finding, observable).
3. A list of domain types that can play the role of this inference-level object.

Thus, the inference-level relations "abstraction" and "specification" both map onto all tuples of two domain relations.

Inference knowledge The inference-knowledge representation consists of three elements. The first element is the declaration of inferences as defined in the expertise model.

```
% inference(Internal name, External name)
% dynamic_role(Inference,     Input/Output,   General name, Specialised name).
% static_role(Inference, , Inference knowledge).

inference(     abstract,  'Abstract').
dynamic_role(  abstract,  input(1),       finding,     'Specific finding').
dynamic_role(  abstract,  output,         finding,     'General finding').
static_role(a  bstract,  relation(abstraction, finding, finding)).

inference(     specify,   'Specify').
dynamic_role(  specify,   input(1),  finding,     'Finding to be clarified').
dynamic_role(  specify,   output,    observable,  'Dependent observable').
static_role(   specify,   relation(specification, finding, finding)).
```

The static role describes the static domain-knowledge used by the inference.

The second element of the inference-knowledge representation is the definition of an inference function that realises this inference by retrieving the necessary domain knowledge and activating an appropriate inference method:

```
inference_function(abstract, [In], Out) :-
    domain_retrieval(find_all, abstraction, Rules),
    rule_interpreter(Rules, In, Out, forward, single_pass, find_one).

inference_function(specify, [In], Out) :-
    domain_retrieval(find_all, specification, Rules),
    rule_interpreter(Rules, In, Out, backward, multi_pass, find_one).
```

The predicate *domain_retrieval* represents a knowledge-base access function that retrieves the required domain knowledge by interpreting the domain-view description of the inference. The predicate *rule_interpreter* represents a call to a production-system technique, where the last three arguments indicate the required control regime in a similar way as is NEOMYCIN:

- *forward/backward* Derive conclusion from the premise of a rule (forward) or set up premise as a goal to achieve the conclusion of the rule (backward).
- *single/multi pass* Stop once a conclusion or a goal has been inferred (single pass) or invoke the rule interpreter recursively (multi pass).
- *find one/all* Stop evaluation (find one) or continue evaluation (find all) of the rule set when a conclusion or a goal has been found.

This production-system technique interprets a domain-relation tuple as $< premise, conclusion >$.

The inference *abstract* is thus realised by a single, data-driven, evaluation of the rule interpreter using the tuples of the two domain relations as rule set. The inference *specify* is realised by a recursive, goal-directed, execution of the same rule set.

Fig. 10.3 shows a a schematic view of a sample execution of the *specify* inference. The input is a finding. The corresponding inference function calls a domain-retrieval function to retrieve all specification rules. This retrieval function uses the domain index to collect

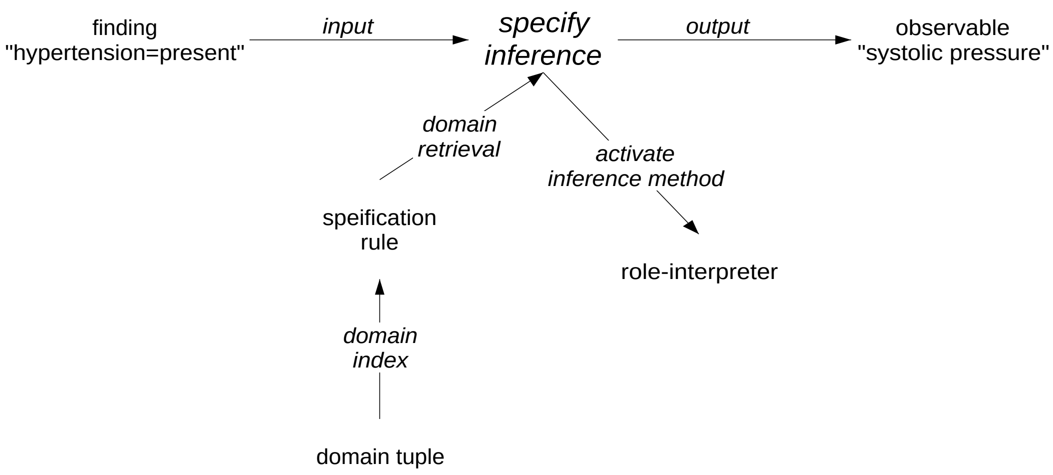


Figure 10.3: Schematic overview of a sample execution of the inference *specify*.

the appropriate domain knowledge. Finally, the inference method (the rule interpreter) is activated which produces the output (in this case an observable).

The third element of the inference-knowledge representation is the actual definition of the inference method. The corresponding Prolog predicates implementing the rule interpreter can be found in Appendix B.

10.3.2 Task knowledge

Given the set of tasks specified in the conceptual model (consisting of both problem solving tasks and of transfer tasks) the designer has to make two – interrelated – decisions, namely:

1. The choice of a control technique for executing tasks. The simplest solution would be to define an interpreter for a representation of the control structures in the expertise model. This solution is sufficient in the case where the communication model does not impose additional requirements on the control regime (see the discussion earlier on HCI functions). This is the solution taken in the diagnostic system described in Sec. 10.4.

   If the communication model imposes additional control requirements, it is usually appropriate to employ a more complex control technique, that integrates the execution of control structures and the control of HCI functions as defined in the communication model. Control techniques that could be used for achieving this are an agenda mechanism, a blackboard or a skeletal planning technique. The Stat-Cons system is an example KBS where the user interface requires complex control [de Greef *et al.*, 1988].

2. The choice of how to represent and update the run-time data. These data can be viewed as the "working memory" of the KBS. This working memory contains the data that are manipulated by the tasks and the inferences: e.g. the current state of the differential, the findings, etcetera. The control terms and the dynamic knowledge roles specified in the conceptual model form the basis for the representation

of working memory: they often reappear in the final system as labels for (sets of) working memory elements.

Most existing KBS's use a simple monotonic technique for updating working memory. We expect that in the next generation KBS's more complex techniques such as truth-maintenance techniques will be used more often. Note that the use of such a technique can pose additional requirements on the output produced by computational techniques realising primitive inferences. An example of such an additionally required output is the "justification" used in an ATMS [de Kleer, 1986].

In Appendix B an example control structure representation can be found for two tasks that apply respectively the *abstract* and the *specify* inference.

10.3.3 Strategic knowledge

Most conceptual models that have been constructed do not contain much strategic knowledge, if any at all. In most systems the strategic part has been "compiled out" into fixed task decompositions with possibly a few strategic decision points that can be influenced by the user (cf. [de Greef *et al.*, 1987]).

If more elaborate strategic knowledge is present, the following techniques could be applicable:

- A production system containing a set of meta-rules with states of working memory as conditions and task activations and/or changes to working memory (e.g. assumptions) as actions.
- An extended blackboard technique such as the "Blackboard Control Architecture" [Hayes-Roth, 1985], where the scheduling part represents the strategic knowledge.

From the viewpoint of the skeletal architecture in Fig. 10.2, these techniques involve the development of a complex central control unit (the term "task interpreter" is probably not appropriate anymore within this context).

Strategic knowledge as meta-knowledge An alternative route is to view the strategic knowledge as a separate meta-system, The PDP system is an example of this approach [Jansweijer *et al.*, 1986]. This approach has been the focus of the REFLECT project [van Harmelen *et al.*, 1992]. In the REFLECT approach the strategic knowledge is viewed as a *meta-theory* about the three other knowledge categories in the expertise model (the object-theory). This meta theory can be described with the same modelling framework as the object-theory. In architectural terms, this means that the skeletal architecture sketched in Fig. 10.2 is extended to a meta-level architecture in which a strategic meta-system reasons about and acts upon a model of the object system. This object-model is causally connected [Maes, 1987] to the corresponding constructs in the actual object system. In practice, this causal connection can only be achieved if the object-system has been can be operationalised in a structure-preserving way.

Note that the architecture of the object-system itself is also of a meta-level nature: the inference-level components reason about an d act upon the (object-level) domain knowledge (see the discussion in Sec. 10.2.1).

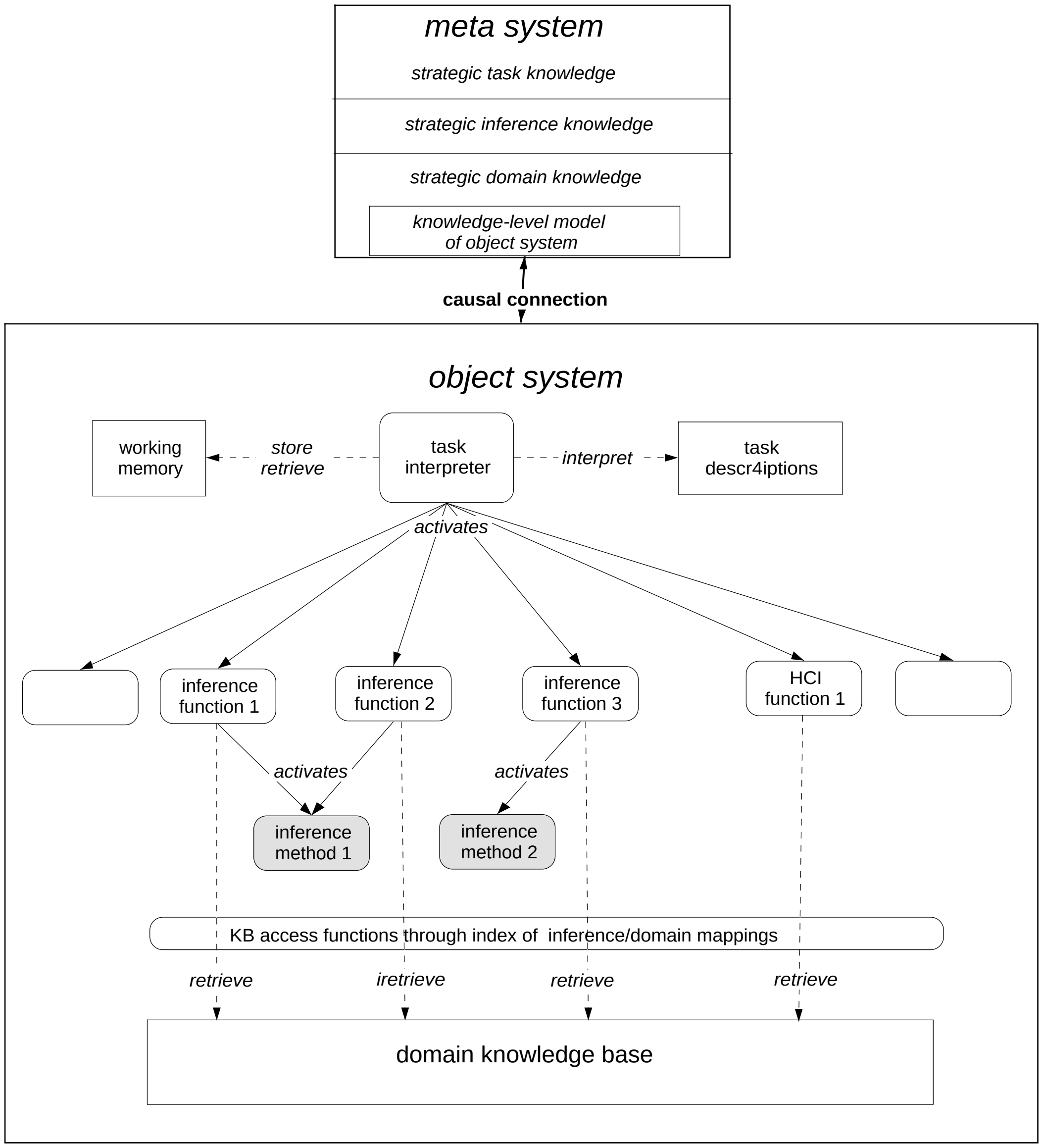


Figure 10.4: Skeletal meta-level architecture in which the strategic knowledge is operationalised through an separate meta system. This meta system uses a a causally connected knowledge-level description of the object system to reason about and act upon this system.

The term "knowledge-level reflection" has been coined for the REFLECT approach to distinguish it from most other meta-level approaches that reason directly about the actual code fragments of a system. The REFLECT architecture allows to build systems that carry out reflective tasks like competence assessment and competence improvement in a flexible way. For more details about this architecture, the reader is referred to [Reinders *et al.*, 1991, Schreiber *et al.*, 1991b, van Harmelen *et al.*, 1992].

10.4 Maximising Support and Flexibility: An Example

Ideally, one would want to combine the support provided by reusable pieces of code and the flexibility offered by the "knowledge-level" programming languages. The grain size of task-specific shells is too coarse: these constitute an implementation of a complete interpretation model. On the other hand, the programming languages do not provide any reusable code fragments such as ready-to-use implementations of inferences. What appears to be needed is an environment with a library of reusable modules that can be used to operationalise elements of a knowledge-level model and allow the knowledge engineer to configure these into a system that meets the demands of the application at hand.

As a prelude to work in task 1.4, we have developed a prototype environment that can be considered as a first attempt in this direction. It contains a library of modules that can be used to operationalise inferences for a class of diagnostic tasks. These inferences appear in the interpretation model for systematic diagnosis. This model represents a method for diagnosis in which a device is examined in a systematic, top-down, manner to find a component that behaves abnormally. Systematic diagnosis can be seen as a form of generate and test. Hypotheses are generated through decomposition of a hierarchical device model into sub-components. A component is tested by predicting the value of an observable using a model of the normal behaviour of the device and comparing this norm value with the observed value. The control knowledge typically has a recursive structure: decomposition is carried out until a non-aggregate abnormal component is found.

This basic version of the model of systematic diagnosis can be extended in a number of ways. These extensions are adaptations of the model that could be necessary in a particular application. They modify the basic model of systematic diagnosis and define in fact a space of potential systematic diagnosis models. Two example extensions are (i) the introduction of complex tests that require system reconfiguration (e.g. reconnecting cables), and (ii) the use of multiple device models each representing a different view on the device (e.g. functional, physical). An overview of the complete set of inferences is given in Table 10.2.

The environment we developed contains modules for all potential inferences, both for the basic version and for those required by the extensions. Each module represents a computational technique for realising the inference. Each computational technique has particular domain knowledge requirements, e.g. the technique for realising the inference in which a norm value is predicted requires a causal model of the normal behaviour of the device.

The environment offers the following facilities to a knowledge engineer who wants to build a system for a particular instance of systematic diagnosis:

- The instantiation of techniques for the required set of inferences.

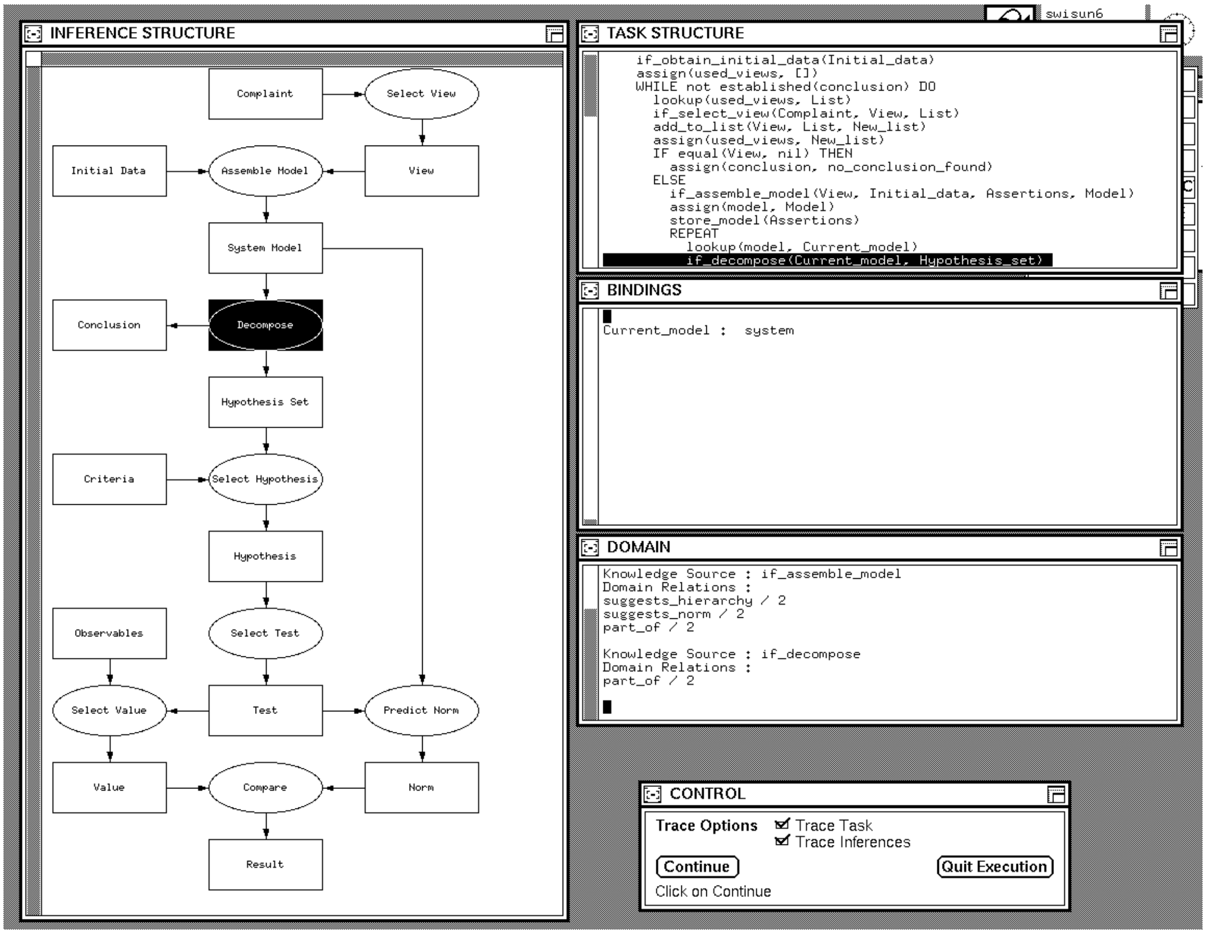


Figure 10.5: Prototype interface for tracing the execution of a system for systematic diagnosis in the vocabulary of the knowledge-level model. The inference structure is shown on the left. The *decompose* inference is currently being executed. The control structure (control knowledge), the bindings (in this case the system model that is currently being decomposed) and the domain knowledge used by the decompose knowledge source (a part-of relation) are shown on the right. The window in the lower-right corner allows the user to trace the reasoning process at the task and/or inference level.

Inference	input	output	description	used in
select	complaint	system model	selection of an appropriate device model	basic version
decompose	system model	differential	hypothesis generation though decomposition of the device model	basic version
select first	differential	hypothesis	select a component for testing	basic version
specify	hypothesis	test	find a test for the a component to be tested	basic version
specify	test system model	norm	predict the normal test outcome	basic version
compare	value, norm	truth value	compare the observed and the expected value	basic version
assemble	complaint initial data	system model	dynamic creation of the device model	extension
transform	system model	system model	allow tests that require a reconfiguration of the device	extension
sort	differential criterion	differential	order the hypotheses in the differential	extension
select	complaint	view	allow multiple device models each representing a view	extension

Table 10.2: Inferences in systematic diagnosis models

- A language to define tasks that specify the sequencing of inferences (control knowledge). The control language is an operationalisation of the modelling language used for describing control structures. It contains additional constructs necessary for implementation, such as invocation of user interface functions and storage and inspection of of intermediate results. An example fragment of this language can be seen in the top-right window of Fig. 10.5.
- A simple editor for entering the domain knowledge required by the selected inferences.

The environment also supports the execution of the resulting system and an interface that allows a user to trace the reasoning process in the vocabulary of the knowledge-level model. Fig. 10.5 shows part of this interface. The interface allows the user to trace the reasoning with respect to various aspects of the model, such as:

- the task that is being executed and its internal control structure;
- the inference-structure diagram in which an inference is highlighted when it is being executed;
- the bindings of dynamic knowledge roles such as “system model”.
- the domain knowledge that is used by inferences that are executed (a static knowledge role).

A detailed description of the environment can be found in [Lemmers, 1991]. The architecture of the environment is an instantiation of the skeletal architecture of Fig. 10.2.

Future perspectives for support The environment for systematic diagnosis models discussed in the previous section is only a first step in the direction of flexible support for operationalising knowledge-level models. It has still a number of important limitations:

- For each inference only one computational technique is provided to operationalise it. This is too restrictive. For example, the inference in which a norm value is predicted can currently only be operationalised with a technique that uses a causal model of the device. For some applications other techniques might be more attractive, for example some form of qualitative reasoning. Ideally, the environment should support a range of techniques for realising some inference.

- The environment provides only limited support for specifying control knowledge. The notion of task decomposition methods, such as used in Components of Expertise [Steels, 1990] can be of value here. Task decomposition methods specify prototypical decompositions of a task into sub-tasks and/or inferences plus information about sequencing these. Incorporating such methods also as reusable constructs in an environment can support the knowledge engineer also in this respect.

- The environment supports the operationalisation of just a small set of models. For example, to be able to cover a large range of diagnostic applications it would be necessary to include also heuristic methods for diagnosis and combinations of systematic and heuristic methods (cf. [Benjamins *et al.*, 1992]).

Other researchers are working along similar lines. Both in Spark/Burn/FireFighter [Klinker *et al.*, 1991] and in PROTEGE-II [Puerta *et al.*, 1991] the aim is to overcome the limitations of the role-limiting methods described earlier by providing the knowledge engineer with a set of predefined computational "mechanisms" from which she can configure a system.

An important research question that arises is whether it it possible to come up with an appropriate typology of such mechanisms. Such a typology would pave the way for building a library of reusable operationalisations of model elements which is not idiosyncratic for one particular approach and can thus be shared by several groups.

10.5 Discussion

The structure-preserving approach as outlined in this chapter has been applied in a number of system development projects. The StatCons system [de Greef *et al.*, 1987, de Greef *et al.*, 1988], developed in an early P1098 experiment, was developed along these lines and served as an important source of ideas for the theoretical background. The mixer-configuration system [Billault, 1989], also part of a P1098 experiment, used the structure-preserving approach and experimented with more flexible forms of control. The Sisyphus system described in [Schreiber, 1992] provides a rigourous example of structure-preserving design of an office-planning problem solver.

The developers of the Fraudwatch application [Killin *et al.*, 1991], implemented in COBOL and in commercial use for some time now, remark that the KADS approach to design in fact leads to a system which is easier to maintain that a conventional system.

This despite the fact that part of the system had to be be implemented in COBOL. Object-systems developed in the REFLECT project where all based on a structure-preserving design, and this was found to facilitate the construction of flexible meta-systems on top of these object-systems [van Harmelen *et al.*, 1992]. Also, the fact that in conventional software engineering researchers are advocating a design approach in which design is seen as adding implementation detail to an analysis model ([Rumbaugh et al., 1991]), supports our view that structure-preserving design is a promising approach.

However, still a lot needs to be done to support designers in the actual process. The support environments discussed earlier are one way of providing support, but are currently only useful in the realm of prototyping. External requirements of the KBS development process are often in conflict with the constraints of such an environment. From our point of view, a promising line of research is to study mappings between knowledge-level models and design languages that guarantee the structure-preserving property for the resulting system.

Acknowledgements Marco Lemmers implemented the systematic-diagnosis system described in Sec. 10.4.

Chapter 11

Configuration of KBS design: theory and tools

Cuno Duursma and Ronald Schrooten

This chapter describes opportunities for a configuration approach to KBS design with *Common*Kads. First an analysis of the overall design process is given in this report. Then the tasks in KBS design most suited to the application of the configuration approach are identified. These tasks are modelled in some depth and various problem solving methods are matched on them. Configuration design by constraint satisfaction is chosen for implementation in a design configuration support tool. The last chapter of this deliverable contains an evaluation of the tool and assessment of the approach.

11.1 Support for KBS design

This section presents an analysis of the KBS design task in *Common*Kads. A short overview is given of different models of design: design as a function and design as a task. Then we proceed with investigating opportunities for KBS support in KBS design. One task, high level design, is selected and elaborated.

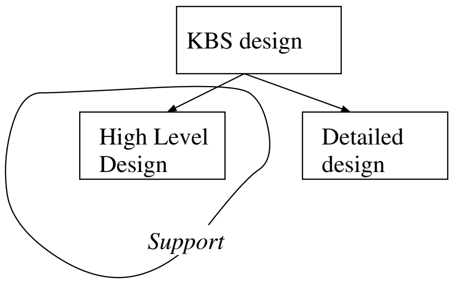


Figure 11.1: *This section discusses models of KBS design.*

11.1.1 Introduction

Design in general is an open ended, creative activity [Smithers, 1992]. A lot of variables have to be dealt with, many options are open and few solutions lead to satisfactory solutions. In [Bernaras, 1993] several models and views of design are discussed. Bernaras describes design as: " Design is a task in which given some needs and desires stated by a customer (or customers) a design solution is constructed such that, when it is realised, the needs and desires of the customer are satisfied". Bernaras distinguishes two approaches to modelling design; 'design as a knowledge process' and 'design as a task' . The first approach has the aim to understand the type of decisions that are being made during the design process. This approach studies design from ill-defined requirements and complex situations [Smithers et al., 1990]. It does however not offer predictive models for design decisions . The 'design as a task' approach tries to find *prescriptive* models for the the activities performed during the design process [Chandrasekaran, 1990]. This approach focuses on well defined problems only. [Bernaras, 1993] has integrated these two approaches by the use of the problem-space concept. Though this integration makes it easier to map the different views onto each other, it does not give the knowledge process oriented models more predictive strength.

In order to support the KBS designer with a KBS we have to be able to formulate a predictive model for at least parts of this task. Unfortunately, KBS design as a whole is too complex and often too ill-defined. In order to be effectively solvable a design task should belong to the 'class-3' design tasks category:

> This is relatively routine design, where effective problem decompositions are known, compiled design plans for the component problems are known, and actions to deal with failure of design solutions are also explicitly known [Chandrasekaran, 1990][page 34].

In order to find areas in design where this condition can be satisfied, we made an analysis of the design process and investigated support in different phases and sub-tasks of the design process. For instance, the designer can be helped by organising the information in such way that the 'cognitive overload' of design variables and requirement constraints is reduced. Also, the designer can be supported by using experience from previous design solutions, or checking his or her designs for intrinsic constraints, or supporting quick 'what-if' evaluations.

In *Common*Kads a KBS design is documented within the structure of the design model. This model provides among others a language to express decisions made in high level design process of KBS. Design decisions depend on the contents of *Common*Kads analysis models and intrinsic design constraints. The focus of this research reported is on a theory that predicts expert-designer's choices and its implementation in a support tool. In order to achieve this, we make a requirement analysis, a conceptual model and design of a design-supporting system for KBS.

11.1.2 Current design practice in *Common*Kads

The top level task in KBS design is the selection and configuration of design components that will be used to create a system with required functionality. The input of this task is distributed over five *Common*Kads models; task model, agent model, expertise model, communication model and the organisational model.

Figure 11.2 shows a diagram of the dependencies between models in *Common*Kads. The approach to the development of the models is 'risk' driven [Taylor, 1992]; this means that the methodology proposes to work first on those tasks that have the highest risk of giving development problems.Fig. 11.3

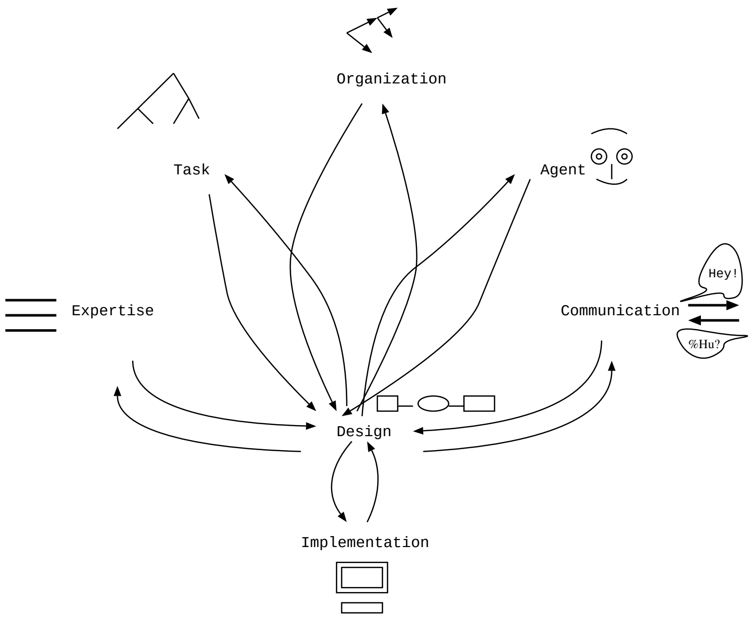


Figure 11.2: *Overview of dependencies between* Common*Kads design model and analysis models.*

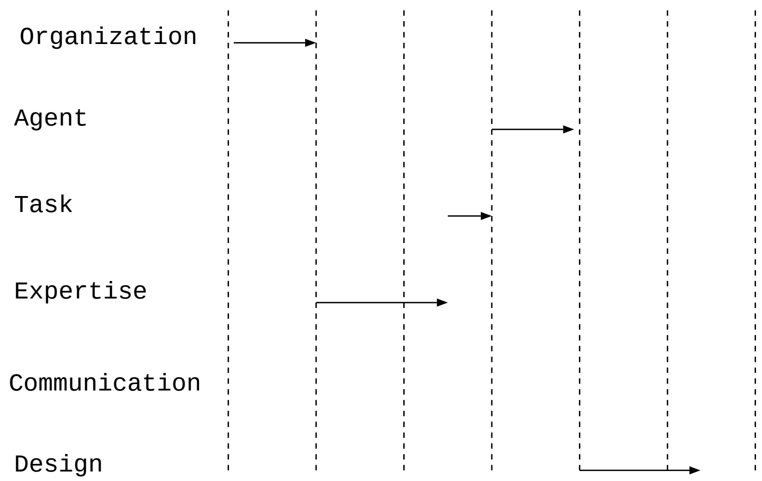


Figure 11.3: *The development of the models proceeds in a risk driven way. If a potential problem area has been identified, the model covering that area is advanced to minimise the chance of failure.*

Example I *Complex interface* In the *Schedule* project [Rademakers, 1991] the acceptability of the complex interface seemed to be the major risk. The interface has the replace large size paper forms, but it had to be seen whether users are able and willing to navigate by means of a visual display unit over the information. Therefore a prototype interface (maquette) of the system was built first. A second result was that the problem solving behaviour of the experts changed one they started to use the interface, because time consuming drawing operations started to be very simple, and some others became more difficult.

Example II *Complex domain knowledge* In the TROTELC project an expert system has been created that diagnoses printed circuit boards [Vanwelkenhuysen, 1991]. In this project major risk of the project was not the interface to the user, but the knowledge representation of the board to be diagnosed. This risk was however only considered when implementation had finished. This proved to be a weak point of the project later.

One of the problems the KBS designer has to deal with, is the number of requirements that must be satisfied. He or she has to integrate all these requirements which are distributed over five different models into one instance of the design model. Moreover, requirements are often contradicting and have to be reconsidered during the design process as no immediate solution can be found.

At some point in the development process, all models should have been developed to such an extend that the requirements between the models can be satisfied. Which models is elaborated first and to what extend during this process is a matter of risks and opportunities.

The following subsections present an overview of the kind of relations between *Common*Kads models and high level design decisions represented in the design model. A *detailed* design is made on the basis of the contents of the TM, EM and CM. High level design decision are made by considering the types of information within these models plus the AM and OM.

11.1.2.1 Requirements on KBS design from the task model

The task model [Duursma et al., 1992a, de Hoog et al., 1993b] "describes the tasks that are executed in the organisational environment where the proposed expert system will be installed in the future". The model can be represented by a (often hierarchically) structured set of frames. Among the attributes represented are input, output and goal of the task.

Several instantiations of the task model can be made during a project, for instance one describing an existing situation, a past situation, or a desired situation. Only the task model that describes the desired situation is relevant as a specification and it serves as an input to the design process.

The task model of the desired situation is a structured set of goals that should be achieved by the system and the user together. An instantiated task model of the desired situation can be part of a of an initial system model, as a description of the functionality of the system to be delivered. During the requirements analysis phase it can be elaborated to a (functional) requirements definition with a communication model. In projects following the traditional 'waterfall' cycle, the Task Model is the starting point for expertise modelling. In projects following the risk driven life cycle this is often the case too, but problems foreseen in expertise modelling could overrule the traditional ordering of phases.

The type of the task in the required TM often guides the selection of the architecture. However, the selection of the architecture also depends on the problem solving method selected in the EM. If the task cannot be matched on any known task, it might be interesting to build a new task specific architecture.

11.1.2.2 Requirements on KBS design from the agent model

The agent model [de Hoog et al., 1993b, Waern et al., 1993] represents the capacities of the various people and systems that play a role in either the existing or the desired situation. The agent model is part of the problem description and can be delivered with the requirements definition.

A second use of the agent model is to check the validity of the task and the communication model. The capacities of the agent should match the capacities needed for the tasks assigned to it in the task model. The ingredients assigned to agents in the communication model should really owned by the agent. Only an agent model describing the desired situation is relevant for the design process.

Some non-functional requirements can be derived from the agent model, like the expertise of the user, the responsibility for the results.

11.1.2.3 Requirements on KBS design from the communication model

The communication model [de Hoog et al., 1993b, Waern and Gala, 1993] describes the communication and cooperation between users and independent system components. It describes all transactions between different agents of the system environment. The communication model thus represents requirements to communication with the future system. The communication model is currently described in a non formal way.

Functional requirements to a system can be described in terms of input output behaviour and completed with a set of non-functional requirements. The communication model is therefore an important source of functional requirements. The basic information of the communication model is what is transferred between the various agents, and who has the initiative for the transfer.

The level of detail for the requirements specifications can vary. It is possible to describe the communication until the very last mouse-click, but it is also possible to describe the communication at a higher level of abstraction, with just a reference to the specific communication metaphor to be applied, like for instance direct manipulation.

In [de Greef, 1989] a communication model has been introduced that represents some communication requirements a way structured around tasks, ingredients and initiatives. This is also how the *Common*Kads model of communication has been shaped. Though the models give some structure and focus to the investigation of the functional requirements; the formulation of the requirements if still a creative task; that can only be done successfully by a team that knows about the user needs and the technological possibilities and costs.

Several requirements to the high level design can be derived from the communication model. For instance, the initiative type and the interaction of the user, the kind of data and knowledge manipulation and the types of explanation the user requires.

11.1.2.4 Requirements on KBS design from the expertise model

Expertise models can be created during the analysis and design phases of a project. During analysis it is used to get an overview of the reasoning and the domain knowledge applied to the task. In the design phase it is used to make a high level and later also a detailed design of the inferencing knowledge and domain knowledge of the system.

The expertise model [de Hoog et al., 1993b, Wielinga et al., 1993] refines the functional specification of the tasks assigned to the system by identifying a an approach to solve the tasks stated in the task model.

While the task model shows the input/output relations required in the system and the communication model the required communication between the system and the rest of the world; the expertise model represents a first step to how these I/O pairs can be achieved. The task and the communication model reflect behavioural views on the system; the expertise model provides an analysis of how the behaviour can be achieved in a 'rational way'.

Several requirements to the high level design can be derived from the expertise model. For instance, the type of problem solving method is related to the type of architecture.

11.1.2.5 Requirements on KBS design from the organisation model

The organisation model [de Hoog et al., 1993b, de Hoog et al., 1993] represents knowledge about organisational aspects related to KBS development. Some requirements to the high level design can be derived from the role of the system in the organisation, like the environment in which the KBS will be deployed, maintenance requirements related to the stability of the organisation, organisation policies for hardware and software, etc.

11.1.2.6 Summary of requirements

- *Task Model*
  1. The required functionality of the KBS in terms of a natural process.
- *Communication Model*
  1. Initiative type for the tasks.
  2. Initiative type for the transfer of ingredients.
  3. Explanation type
- *Expertise Model*
  1. Problem Solving Method
  2. Assumptions
  3. Domain Ontology
- *Agent Model*
  1. Agent type
- *Organisation Model*

1. Environment
2. Maintenance requirements
3. Hardware and software policies

The top level task of a 'typical' software engineer designing a KBS using *Common*Kads can be considered decomposed of several sub-tasks (see the TM worksheet in Fig. 11.4).

11.1.2.7 Required Task Model: design with configuration support

The task model of the current situation is the starting point of our search for opportunities to improve KBS design with *Common*Kads. We have focussed on *high level design*. *Common*Kads specific approaches to *detailed design* are discussed in [Duursma et al., 1992].

We assign the Task to the system to *predict the use of design components from provided requirements in the analysis models.* Secondly, we have to investigate whether a computational realisation of this configuration design process can be made. The order of these activities is not necessarily fixed, but determined by the methodology.

In a structured software engineering process the 'components' are not only the computational programming language constructs. Higher level constructs also play an important role. A good choice of high level design concepts reduces the complexity of the design process, improves the efficiency of the software and also determines the strength of secondary requirements. The theory we are looking for should predict the choices for elements a range of different concepts. These concepts are listed in the *Common*Kads DM; they include:

1. *Application specification decomposition.* The way the tasks from the TM are distributed over separate specifications. Different application specifications can use different specification languages. This can be useful when they specify information of different nature, like interfacing and reasoning tasks.
2. *Application specification language.* The selected language to specify the application. This can either be a language with formal semantics, an operational language, or a conceptual language.
3. *Application specification.* The specification of the required behaviour in terms of the application specification language.
4. *Detailed design.* The detailed design of the application specification in terms of the architecture.
5. *Architecture.* The choice, or design, of the architecture used for the application.
6. *the Detailed application Design.* The design of the application specification on top of the selected architecture(s).

If a suitable architecture is not available, then it needs to be designed. Architecture design can be modelled in an additional layer using the same ontology; the task refers in this case to the design of the architecture, the application specification the specification of the functionality of the architecture. This can be repeated a number of times to create several means-ends layers. The DM comes 'standard' with terminology where the application is realised by two means-ends levels, first the application design on the architecture, and secondly architecture design on top of the target language.

TM

Continuation of: *None*

***Common* Kads Task Model Worksheet**

Task Model id: *Design .1*

Management

Model Type:	[x] existing [ ] target: [ ] current at: [ ] other:.........................	*(tick one)*
Situation:	*current*	*(the situation in terms of states of models or text)*
Objective	*model 'today' practice in KBS design.*	*(the objective of making this model, in terms of situation or states of models)*
Risk	*unclear where best support to KBS design should be given.*	*(the risk you want to avoid)*
Rationale:	*The configuration support system should be able to help with time consuming design tasks for which we have the theories and technology available to solve it.*	*(why you think this will avoid it*
Target State		*(a model state of a type of task model)*
Decomposition approach:	[ ] hierarchical [ ] clustering [x] model-based *(design model)* [ ] other	*(tick one)*

Task Decomposition

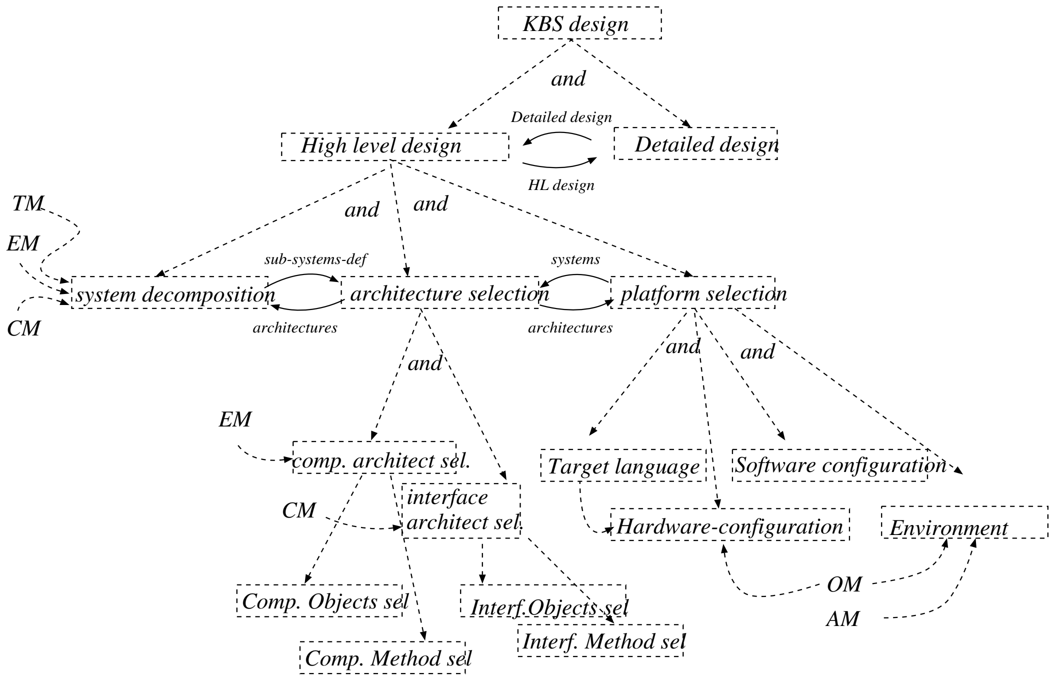


Figure 11.4: *Decomposition relation and ingredients transfer in the task model that represents of the* KBS *design task without* KBS *support.*

A theory for design should preferably predict accurate values for all of the slots in the design model: the decomposition of the application specification, the application specification itself, the architecture, the detailed architecture, the selection of the target language, and the detailed architecture design on top of this target language. Whether such a theory can be developed depends on our insight in the design process and on the observables that can be used for the prediction.

TM	Continuation of: *None*	***Common* Kads Task Model Worksheet**	Task Model id: *Design .1*

Management

Model Type: ☐ existing ☒ target: ☐ current at: ☐ other:......................... *(tick one)*

Situation: *required* *(the situation in terms of states of models or text)*

Objective *model KADS KBS design with configuration support.* *(the objective of making this model, in terms of situation or states of models)*

Risk *Set the scope of the configuration tool support.* *(the risk you want to avoid)*

Rationale: *The configuration support system should be able to help with time consuming design tasks for which we have the theories and technology available to solve it.* *(why you think this will avoid it)*

Target State *(a model state of a type of task model)*

Decomposition approach: ☒ hierarchical ☐ clustering ☒ model-based ☐ other *(tick one)*

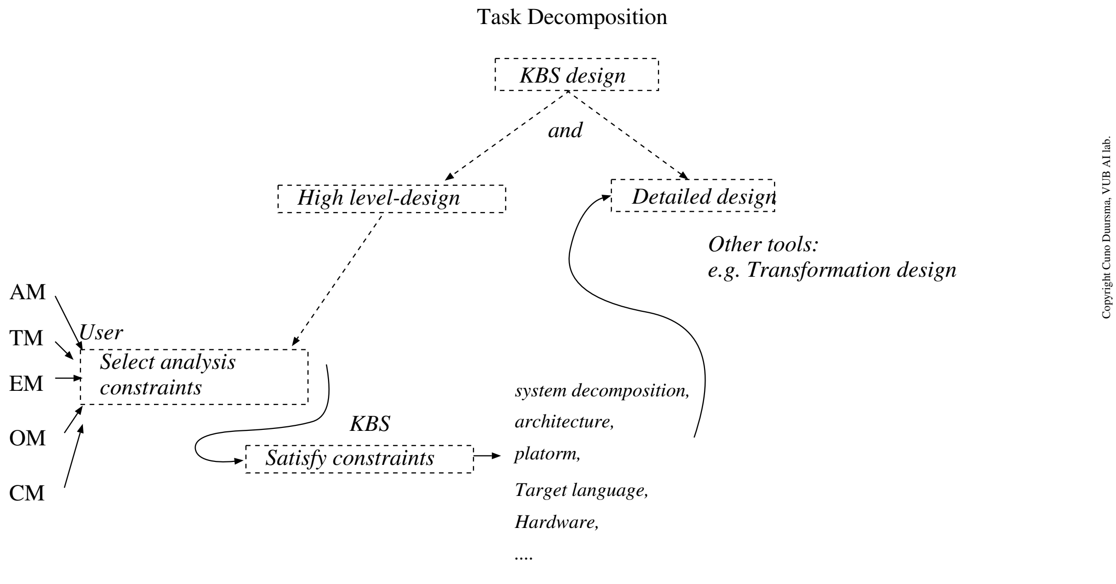


Figure 11.5: *Required task model: Design with configuration support.*

11.2 Theories for high level design

In this section we apply different problem solving methods on the high level design task. For each of these methods we investigate the resulting model ontology requirements and -

assumptions. The results are compared and on this basis constraint satisfaction is selected and elaborated upon

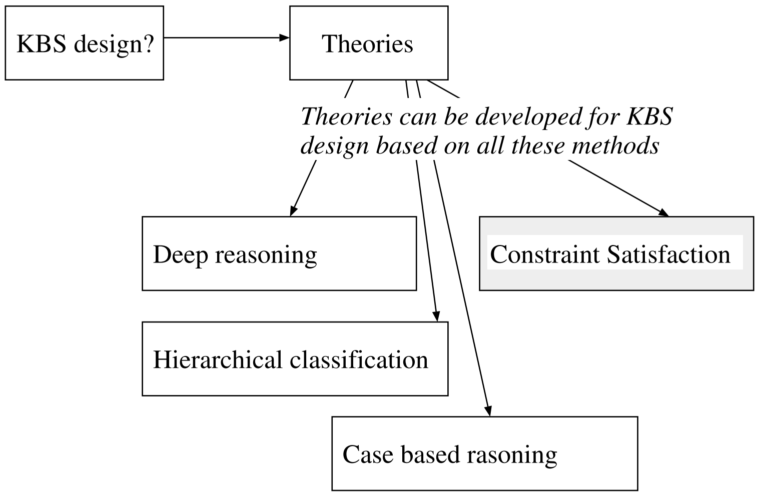


Figure 11.6: *This section discusses different methods that can be used to formulated a design theory.*

11.2.1 Introduction

A KBS can perform a task if a predictive theory is available or can be developed about the task. In this section we analyse the process of component-selection in high level design and show that a predictive theory for the selection of design components can be formulated.

- *Natural process:* the selection of design components as specified in the *Common*Kads DM. These components range from pre-existing system decompositions, architectures, fragments of architecture specification, pieces of application specification, or combinations of those.
- *Experiment:* What design model configurations are chosen by an expert KBS designer for a set of *Common*Kads analysis models.

The high level design theory should predict the use of design components from a configuration of analysis models and constraints between design elements. The theory should associate problem situation types with a selection and configuration of design elements. The associations should be bi-directional in order to make them also useful for predicting the constraints on the requirements starting from the configuration of the design.

11.2.2 High level design approaches

The following few sections introduce different approaches for formulating, formalising or operationalising high level design. The approaches are not completely orthogonal. They are all "generic components" in the sense of the *Common*Kads library [Breuker and Van de Velde, 1993]: components that can be used to construct a predictive model. For instance, high level design can be mapped on the generic task "configuration". Configuration can be performed by classification, by case based reasoning and also by constraint satisfaction. High level design can also be mapped directly onto these components without the mapping to a configuration task. The result is for instance high level design by case based reasoning or high level design by classification. However, if the mapping to a 'generic component' is made via a generic task available in the *Common*Kads library, then the library's model ontology translation tables can be used.

We first interpret high level design in terms of the generic configuration task. The next subsection interprets high level design as a classification task. After that we try a design approach based on first principles. Finally we present high level design as configuration via constraint satisfaction.

11.2.2.1 High level design as a configuration task

Configuration design is refers to designs where the problem is to find values for a set of parameters that satisfy a number of constraints. More specifically:

> Parametric design can be characterised as a kind of design where a set $E = e_1, e_2, \ldots e_n$ (elements), $R = r_1(e_1, e_2, \cdots e_n), r_2(e_1, e_2, \cdots e_j), \cdots$ (relations) and $S =$ (structure) are given (...). Constructing a solution in parametric design is the process of instantiating the values of the attributes of E in S such that R holds.

(Adapted from [Bernaras, 1993])

Examples of configuration task are the office assignment (SISYPUS '92) and the elevator design problem (SISYPUS '93 [Yost, 1992, Gaines, 1994]).

We take these constraints as a starting point and develop a theory that can predict good design configurations for KBS. There are several alternative directions in which the configuration theory can be developed from this starting point. The different alternatives have different model ontology requirements [Duursma et al., 1993], different consilience [Duursma, 1993a], simplicity and make different assumptions.

The detailed design of a KBS is not a configuration task: the variables are not known in advance, all constraints cannot be enumerated nor can all possible values if the variables be enumerated. There is simply too much freedom in the number and the range of the solution variables. This makes the search space for the applicable configurations infinitely large. For the other choices of the design the dimensions of the solution are known, and less alternatives available. Design choices can also be associated with categories of requirements specifications.

11.2.2.2 High level design via a classification method

The simplest approach to high level design is to classify the problem case into a known and previously solved category; retrieve the solutions for that category and present these results as the prediction of the theory. In its most simple form classification is just a table lookup kind of theory. Usually these categories are organised hierarchically. In the Clancey's model [Clancey, 1985] the class hierarchy has been split into two distinct hierarchies linked with matching relations. One hierarchy contains classes representing various problem categories, the other hierarchy represents solution categories. Elaboration of the configuration theory by hierarchical classification results into the following model ontology requirements for the configuration theory:

Classification	Configuration design
problem class	specification type
problem class feature	feature of the specification; an aspect of the configuration of a number of *Common*Kads analysis models
matching relation	specification type – design solution type relation
problem-class – problem super-class relation	General problem-statement – specialised problem-statement relation.
solution-class – solution super-class relation	General design solution – refined design solution relation

Hierarchical classification can be performed in several distinct ways, see for instance [Breuker and Van de Velde, 1993] for examples of the application of four different role limiting methods applied to classification.

Though the structure of a theory based on classification is straightforward, the model ontology requirements listed above cannot easily be satisfied for configuration design. For configuration of *Common*Kads analysis models the relevant design model structure needs be listed. If the design model has n different parameters ranging over an average of v different values; then v^n solution classes are required. In the same way, if these solutions are required for i parameters in the *Common*Kads analysis models ranging over an average of w values, then a total of w^n problem classes are required.

These results make configuration design by classification feasible only for high level design: situations where a limited number of aspects of the five analysis models can be used to set some parameters in the design model. Knowledge acquisition for a theory of detailed configuration design by classification seems unfeasible.

11.2.2.3 High level design via a deep reasoning method

Another possibility is to find underlying design principles that make one configuration of the design model more successful for a set of *Common*Kads analysis models in a certain case than another configuration. This approach could be based on a theory how design choices affect the primary and secondary design requirements. It is not easy to describe how, for instance, the choice for a specific architecture affects the maintainability of a classification task starting from first design principles.

This approach requires a set of general design principles that predict the configuration of the design model from the configuration of the analysis models. This is feasible for detailed specifications. A theorem prover is actually a predictive design theory able to produce operational specifications from formal specifications.

It is difficult to formulate general principles to predict the values of parameters representing higher level design choices, like architecture and different application decompositions. However, the *structure preserving* design approach described in [Duursma et al., 1992], is based on the design principle of structure preservation. This simple but powerful design principle predicts values for higher level design parameters like application decomposition and part of the detailed application specification design. Though quite powerful, this single principle does not make valid predictions in cases where:

(1) different, contradictory requirements with different structure are used in the models. (2) An architecture is applied that is different from the language used to formulate the requirements.

Principles covering these difficult cases would have to take more detailed information of the analysis models into account. For instance, both classification task and the black-board architecture would have to be described at much finer degrees of detail; like the ordering of data, the distribution of functionality, the match of the classification on the function decomposition on the black-board. Only then basic design principles like *maximal coherency* and *minimal dependency* could be used to configure the design model.

11.2.2.4 High level design via a case based approach

Extrapolation from similar cases is similar to the classification approach, but instead of abstracting in advance, the cases are kept and abstracted and matched only when they are required for solving a new case.

Model ontology requirements (called vocabulary) of this approach are[1]:

- *Case:* Refers to a combination of description of an analysis models and a configuration of the design solution that worked in the past.
- *Similarity:* A measure to express the difference between the new case and the old case. This similarity measure is difficult to develop for qualitative cases like KBS design.
- *Extrapolation function:* The method used to estimate modifications to the available case solution from differences between the available case problem and the new problem.

The case based approach can be based on the higher level classifications or on basic design features: this is determined by the chosen indexing features.

11.2.2.5 High level design via a constraint satisfaction method

Another approach to elaborate the design theory is to focus on the constraints a choice of one element of a solution for a problem makes on the other elements that can be selected. It is a kind of jig-saw approach to problem solving; the selection of one element of the puzzle sets requirements for the neighbouring parts; and these set requirements again for their respective neighbours again, until the whole board has been covered.

Like the case-based-approach, constraint satisfaction can be used on deeper principles or on compiled knowledge. The model ontology requirements of the configuration design by constraint satisfaction are as follows:

The constraint satisfaction approach does not work by abstracting problem cases to abstract problem descriptions and associating these with abstract and more refined solutions. Instead, constraints between the problem and the solution are represented. For the configuration approach this means that the specifications constrain the possible values for parameters in the design model.

[1] See [Duursma, 1992b] for a discussion on the model ontology requirements of 'extrapolate from a similar case'.

Constraint satisfaction	**Configuration design**
variable	specification parameter, design model parameter
domain	list of possible values of each specification and design model parameter
constraint	list of tuples of requirements parameter values and design parameter values.

11.2.2.6 Evaluation of alternative methods

It is not possible to develop a configuration theory for design with 'easy' model ontology requirements, high consilience and simplicity and limited computational complexity. The design task is simply too open ended: the complexity of the design task is too high.

The different alternatives discussed pose different opportunities and problems for the elaboration of the design theory. Elaborating the theory as a classification problem requires the identification of a lot of problem solution classes. An approach following basic designers principles requires general and widely applicable principles. These are only partly available. The case based approach and the constraint satisfaction approach seem feasible if cases or constraints are chosen at a suitable level of detail.

11.3 Configuration design via constraint satisfaction

This section shows the analysis and specification of the hierarchical constraint satisfaction method for Operationalisation of the generic parametric design task. As the high level design task has been described in terms of this generic task in the previous section, this will produce an Operationalisation of the high level design task.

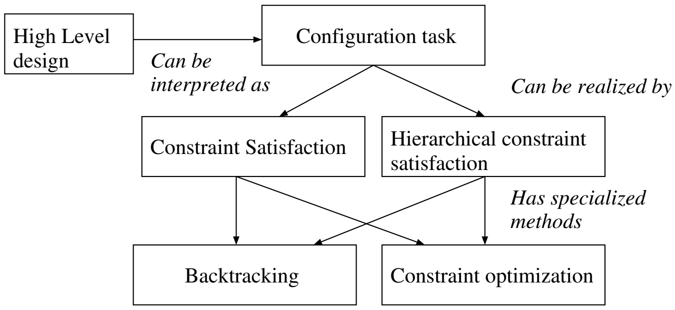


Figure 11.7: *This section discusses how constraint satisfaction can be realised and how can it be mapped on configuration design.*

11.3.1 Introduction

Constraint satisfaction refers to a range of AI techniques and systems with algorithms satisfy relations in a network of variables. Different approaches have been developed, like backtracking, constraint propagation and hierarchical constraint satisfaction. Besides these techniques, the constraint approach is also important for is model ontology requirement (see [Duursma, 1992a]): to analyse a problem in terms of variables and constraints between these variables. A range of problems can be represented naturally as constraint satisfaction problems, for instance parametric design tasks and scheduling tasks.

Two classes of constraint satisfaction problems (CSP) can be distinguished: Boolean CSP's and constrained optimisation problems.

11.3.1.1 Boolean constraint satisfaction problems

A Boolean CSP is characterised as follows [Mackworth, 1987][page 206]

- Given is a set V of n variables $\{v_1, \cdots, v_n\}$.
- Associated with each variable v_i there is a a domain D_i of possible values.
- On some specified subsets of those variables there are constraint relations given, that are subsets of the Cartesian product of the domains of the variables involved.
- The set of all solutions is the is the largest subset of the Cartesian product of all the given variable names such that each n-tuple in that set satisfies all the given constraint relations.

Reporting on the solutions can either consist of the complete set of solutions, of one particular solution, or of the empty set if the SCP is unsatisfiable.

The satisfiability decisions for a CSP is equivalent to determining the truth of a well-formed formula in first-order predicate logic:

$(\exists x_1)(\exists x_1)\cdots(\exists x_n)$
$(x_1 \epsilon D_1)(x_2 \epsilon D_2)\cdots(x_n \epsilon D_n)$
$P_1(x_1) \wedge P_2(x_2)\cdots P_{12}(x_1, x_2) \wedge P_{13}(x_1 x_3) \wedge \cdots \wedge P_{n-1}, P_n(x_{n-1} x_n)$

P_{ij} is only included in the formula above if $i < j$ since it is assumed that the constraints are bi-directional and P is therefore commutative, i.e. $P_{ij}(x_i x_j) = P_{ji}(x_i x_j)$.

Algorithms for Constraint Satisfaction Problems Two types of algorithms have been developed for solving CSP's: backtracking algorithms and consistency algorithms. The first class of algorithms explore the set D by:

1. sequentially instantiating the variables $\{v_1 \cdots v_n\}$,
2. checking the validity of the constraining relations,
3. re-instantiating variables with alternative values if relations cannot be fulfilled.

Consistency algorithms are complementary to the backtracking algorithms. Consistency algorithms have been developed after a careful analysis of thrashing behaviour in backtracking algorithms [Mackworth, 1977]. They can be most easily be explained by the use of the network model:

- For binary constraints each edge in the graph between vertices i and j is replaced by arc (i, j) and an arc (j, i)
- A Node, consisting of a vertex i and the associated domain of variable v_i is *node consistent* if: $\forall(x)[x \epsilon D_i] \supset P_i(x)$

  This means that for each of the nodes, the variable assignment satisfies the local constraints. This does imply not that all constraints have been satisfied.
- The graph is *arc-consistent* if:

  $(\forall x)[x \epsilon D_i] \subset (\exists y)(y \epsilon D_j) \wedge P_{ij}(x, y)$

11.3.1.2 Constraint optimisation problems

Constraint optimisation extends constraint satisfaction with optimisation support. In problems we like to solve different constraints often have different priorities in being realised in the solution. Weights or preferences can be specified in different ways, and can be combined using several schemata. The definition of constraint optimisation is not as clear cut as in boolean constraint satisfaction; and several algorithms give their own interpretation to the problem.

11.3.1.3 Hierarchical Constraint satisfaction

Hierarchical constraint satisfaction (HCS) can be used to improve the representational strength of boolean constraint satisfaction and constraint optimisation by taking advantage of the common structure: in many real-world problems, objects can be grouped into hierarchical classes according to some of their properties. This feature makes hierarchical constraint satisfaction a useful representational device. Isa relations are transitive relations and can therefore be represented by a directed acyclic graph (DAG).

CONTAX [Meyer, 1992] is an example hierarchical constraint satisfaction system. CONTAX offers boolean constraint satisfaction and also, though limited, constraint optimisation. In CONTAX, domains are arranged in isa hierarchies. Constraints can be defined between constraint variables, and sub-variables inherit the constraints of their supers. Inheritance of constraints is formally defined as follows in CONTAX:

Definition **Inheritance of constraints in** CONTAX: *Let* $R_j \subseteq D_1 \times \cdots \times D_i \times D_k$ *be a constraint,* $(d_1, \cdots, d_i, \cdots, d_k)\epsilon Rj$ *be a tuple in* R_j. *Then for all* $\tilde{d}_i \epsilon D_i$ *the tuple* $(d_1, \cdots, \tilde{d}_i, \cdots, d_k)\epsilon R_j$ *if* $\text{ISA}(\tilde{d}_i, d_i)$.

11.3.2 High level Design as a HCSP

The final step is to formulate the high level design problem in terms of hierarchical constraint satisfaction. Once this has been done, the resulting task-, domain- and method-specific ontology can be 'stuffed' with design knowledge.

HCSP required the identification of:

1. The domain hierarchy,
2. The variables,
3. Constraints,
4. and Constraint strength

The *domain hierarchy* is split into two parts; analysis situations on one side and design solutions on the other. On each of these sides several hierarchies can be build representing either analysis or design decisions in abstraction hierarchies linked by ISA links (see figure Fig. 11.8)

The *variables* represent analysis situations and design decisions. Each of the variables ranges over a set or a hierarchy of possible values. The next section will show how design principles are used to choose these variables.

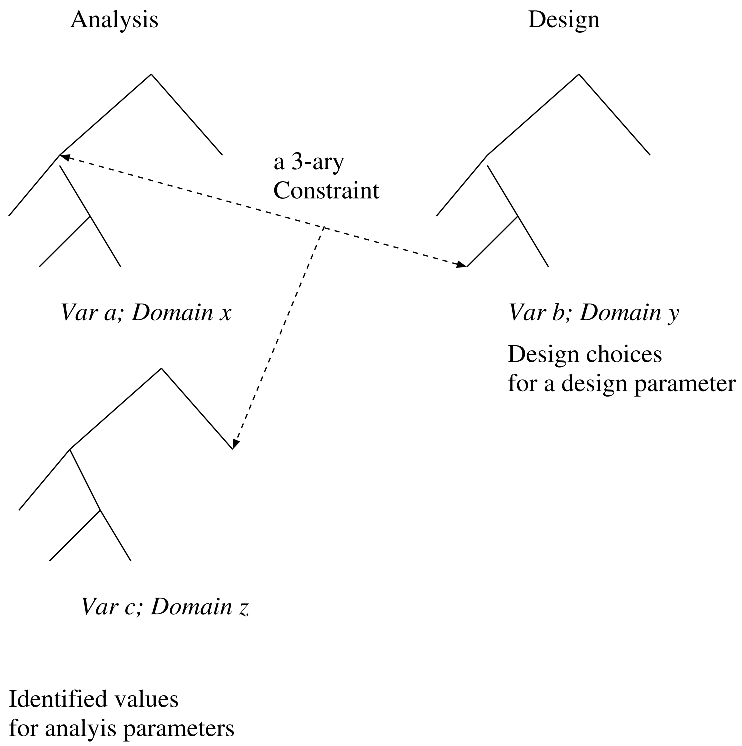


Figure 11.8: *Examples of domain hierarchies in* HCSP *applied to high level design.*

The *constraints* represent constraints between analysis and design situations. They can be of any number higher then zero. Constraints can be represented between different abstraction levels in the domain hierarchies.

The *constraint values* represent the importance of satisfying a constraint. *Hard* constraints must be satisfied for a successful design. *Weak*constraints should preferably be satisfied but are not essential.

11.4 Design Guidelines

This section introduces design guidelines as knowledge that is used in the configuration design process. Examples are provided and the relation with constraint satisfaction is identified. At the end a brief evaluation of the approach is included.

11.4.1 Introduction

Design guidelines form a bridge between the *Common*KADS analysis models and the design model.
A design guideline is a relation between a design component and a recognisable feature in the problem domain. We stress the fact that a design guideline should always be related to a design component. In this approach of configuration design the design components are the reusable building blocks. Recognisable features of the problem domain can be needs, requirements, facts or other specifications in the analysis models. The fact that they are

recognisable depends on whether they can be discovered during analysis or not.
Because of this strong connection with design components we believe that useful design guidelines can only be introduced via investigation of reusable design building blocks. A lot of work in this area is done in *"Designing Expert Systems - A Guide to Selecting Implementation Techniques"* [Kline and Dolins, 1989]. Most of the guidelines that we use are adopted from this book.

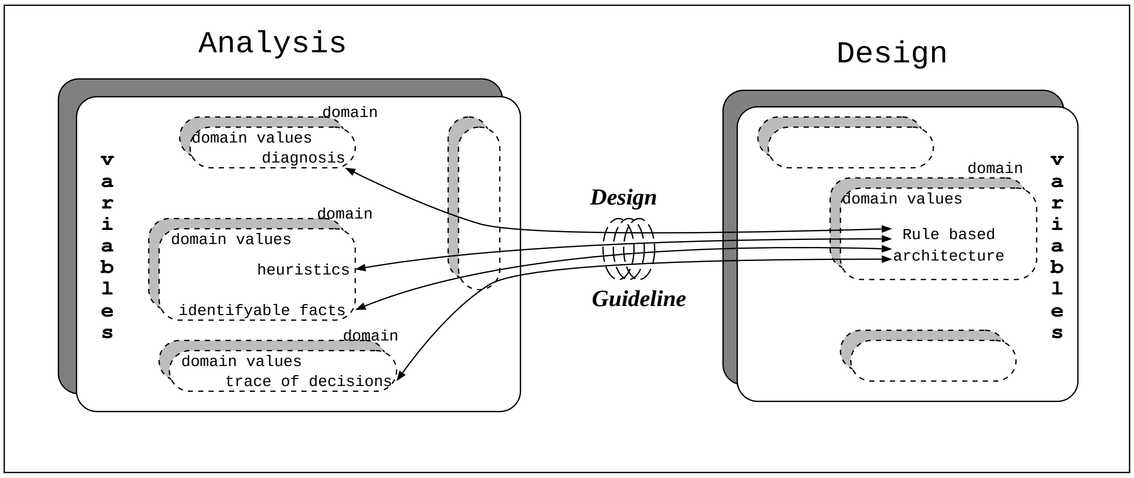


Figure 11.9: *A design guideline is knowledge that is used to bridge the gap between the analysis models and the design model.*

11.4.2 Guidelines as Constraints

In our configuration design approach we implemented the design guidelines as bi-directional constraint relations. In constraint satisfaction terminology this means that we use the guidelines for identifying the constraints between several *domain values* which serve as instances for constraint *variables*. In this way we introduced meaningful knowledge that can be handled by a constraint satisfaction inference mechanism.

11.4.3 Examples

Examples of design guidelines represented as constraint relations between reusable design components and recognisable analysis features.

Guidelines related to a blackboard architecture. First we give a brief description of a blackboard in terms of the Kads design model [Golfinopoulos, 1993] (see also appendix).

Name: Blackboard

Content Description: A blackboard architecture consists of three major components: *knowledge sources* which contain the knowledge needed to solve the problem and can work independent. A *blackboard data structure* that keeps track of the global problem solving state. A *controller* that let the knowledge sources respond to changes in the blackboard. [Velthuijsen, 1992].

Architecture design .

```
Computational:
  Paradigm: Blackboard;
  Command: read; (from the blackboard)
           write; (write on the blackboard)
           match; (monitor the blackboard)
           act; (perform some activities)
           start;(control)
           end; (control)

  Uses: Objects used by the knowledge sources (unit, hypotheses,
                                    elements, attributes, etc.);
  Supports: knowledge sources; (a KS is a an activity, implemented
            as a piece of program, called the KS action part,
            together with a data structure that describes the action part.
```

Design Guidelines:

- There is a constraint relation between the *use of a blackboard data structure to record hypothesis* and the fact that *solving the problem involves transformation between several different levels of description* or *the availability of several diverse knowledge sources*.
  These features can be found through analysis done in the expertise, organisation and agent model.

- There is a constraint relation between the *explanatory capabilities of a blackboard system* and the fact that *explanation should concentrate on accounting for details of the situation under analysis and not on explaining the reasoning*.
  These features can be found through analysis done in the expertise and communication model.

- There is a constraint relation between the *control mechanism of a blackboard architecture* and the fact that *an opportunistic problem-solving strategy is required (i.e. decide dynamically on a problem-solving strategy* or *a planning task type is identified.*
  These features can be found through analysis done in the expertise and task model.

Guidelines related to a rule-interpreter architecture. First we give a brief description of a general rule-interpreter in terms of the Kads design model.

Name: Rule-interpreter

Content Description: A Rule-interpreter architecture uses rules to store knowledge (in most of the cases heuristics derived from an expert). Rules are condition(antecedent)/action(consequent) frames that can be handled by the inference mechanism. Selection of the rules is managed by a control strategy and conflicts are resolved via a conflict resolution strategy. [Bahrami, 1988]

Architecture design .

```
Computational:
  Paradigm: Rule-based;
  Command: rule of inference;
           control strategy;
           Conflict-resolution strategy;
 Uses: Objects used by condition and action parts of the
       rules or the rules themselves;
 Supports: rule of inference: modus ponens, first order logic,
                horn clauses.
           control strategies: forward chaining, backward
                chaining.
           conflict-resolution strategy: rule ordering, size
                ordering, least recently used, most recently used,
                data ordering.
```

Design Guidelines:

- There is a constraint relation between the *shallow reasoning capabilities of a rule-based architecture* and the fact that *the problem type is in the area of diagnoses and can be solved through the use of heuristic that shortcut the causal path of symptoms to conclusions*
  These features can be found through analysis done in the expertise and task model.
- There is a constraint relation between the *ability of representing and reasoning with incomplete knowledge in a rule-based architecture* and the fact that *incomplete knowledge of facts of the problem, limited access to information in this environment so that these uncertainties about facts of the case cannot be resolved simply by asking for the information, important to achieve as complete as possible an understanding of the fact of the case*
  These features can be found through analysis done in the expertise, communication and organisation model.
- There is a constraint relation between the *explanation in a rule-based architecture* and the fact that *it is sufficient to get explanations that tell how the expert system arrived at its answer*
  These features can be found through analysis done in the expertise and communication model.
- There is a constraint relation between the *gathering information in stages through the use of "screening clauses" in rules* and the fact that *certain facts should be established before the expert system request a particular item of information*
  These features can be found through analysis done in the expertise, task and communication model.

11.4.4 Evaluation

11.4.4.1 Advantage

Besides the benefits of guiding a designer towards a reusable design this approach has some extra benefits.

Maintainability : Having the link between needs taken from the analysis models and the actual design of the software can be an improvement over current debugging and maintenance methods [Green et al., 1986]. Design guidelines provide this link. A maintainer can use these guidelines to see if original stated needs and requirements are not violated by maintenance changes in the design of the KBS.

Rational : The design is configured via the use of the design guidelines. These guidelines represent the taken design decisions. A report of all the guidelines that were used to construct the design model can serve as a rational for the design at hand.

Compromising : Design is often stated as a problem of compromising on requirements that are formulated by interest groups (i.e. users, management etc.) [Vanwelkenhuysen, 1993]. A designer can now *plug and play* with the severable compromisable requirements and see to what kind of designs the different combinations result. The inner relations of the different requirements and their impact on the design will become more clear and the several possible designs can be generated and compared very fast. This can be seen as a small tornado in the huge spiral model of the KBS development life cycle Fig. 11.10.

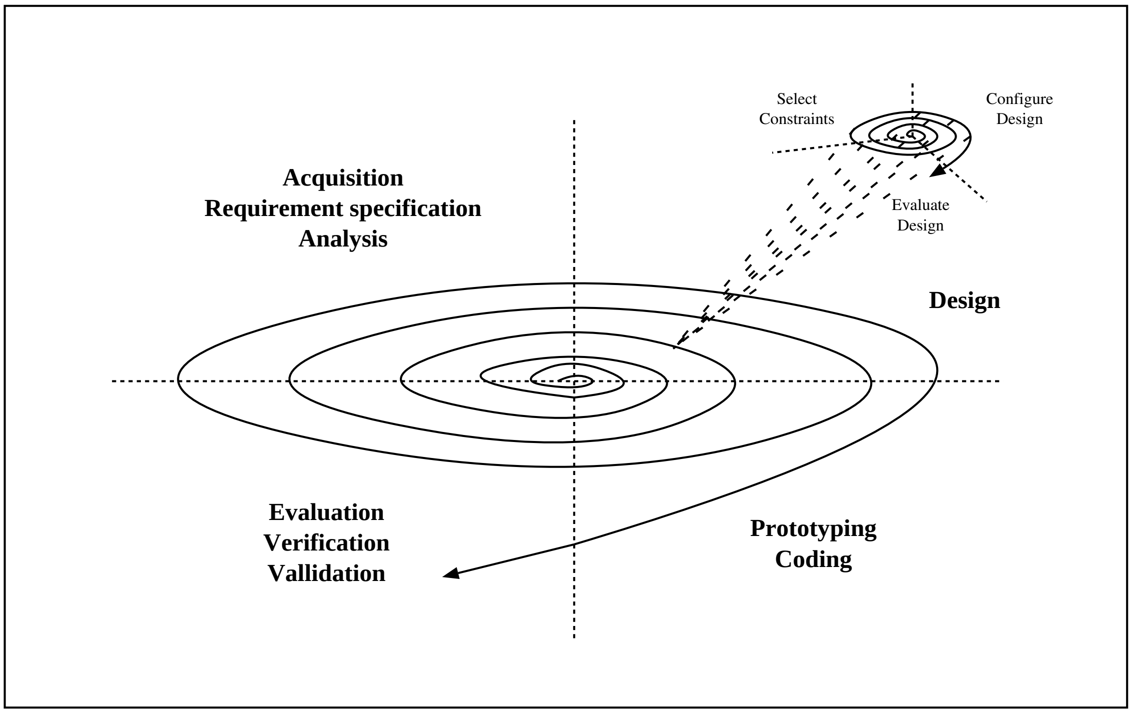


Figure 11.10: *The design process is in itself a spiral development process within the overall spiral model of* KBS *development.*

Validation : Validation is testing the issue of *building the right system.* This reflects on the relation between the *system itself* and with what was *expected* from the system. Design Guidelines provide support for this relation because they make this link very explicit. Furthermore it becomes possible to inspect, motivate or criticise the resulting system on the based of the knowledge that the guidelines represent.

11.4.4.2 Disadvantage

Knowledge Acquisition : Acquiring the correct guidelines for each design component is difficult.

Terminology & Context : Working with reusable component requires a general terminology to identify them. In this guidelines approach we use analysis features. Because there is no fixed terminology we create the problem of mismatching features that have the same meaning but have a different name because they where formulated in a different context.

Theory : A good metric or reliable theory to reduce the set of possible solutions and order them according to quality criteria is not included.

11.5 A Configuration Design Tool

A configuration design tool is implemented to test our theoretical ideas. Specifications for configuration design are made explicit. The design and use of this tool is described. Finally an evaluation of the prototype is given.

11.5.1 Introduction

Before describing the functionality [2] of the tool we first will situate it according to other existing tools. This identification and classification of the tool will tell us something about the range of functionalities it should provide.

According to Forte and McCulley [Forte and McCulley, 1991] tools can be classified into two main classes: horizontal and vertical tools. Horizontal tools are used throughout the entire software process (like the *Common*KADS Workbench). Vertical tools are used in a specific phase of the life-cycle or activity. The configuration design tool can be classified as a vertical tool which is restricted to translating analysis into the design of a KBS implementation. It is useful to know this since it narrows the application domain of the tool. In this case a small restricted part in the overall development process of a KBS.

Within a recently developed classification scheme for CASE technology [Fuggetta, 1993] we can situate our tool in the category of *Configuration management tools*. This tells us for instance that the tool should be able to do *item identification*, *adaptation of items to construct a configuration* and *control on the impact of changes on items*. Items are in this case the reusable design components. The identification is done through the use of guidelines, adaptation is provided via the *Common*KADS design model framework and the control on impact is realised via the design guidelines as well as by the internal relations in the design model framework.

[2] A complete specification of the tool together with the configuration design approach is described in [Schrooten and Duursma, 1994].

11.5.2 High Level Design

Application Configuration Design

Name: Configuration Design Tool

Type: Knowledge based

Functionality: The configuration design tool is a design aid tool that provides computer support for finding and structuring the software design of a KBS to be built. To do this, it interlinks reusable design building blocks which come from a library and provides a convenient graphical interface. The reusable designs are indexed through the use of design guidelines. A reusable design component is selected by means of a constraint network that is based on these design guidelines. The design guidelines form the back-bone knowledge of the complete configuration design process. The constraint satisfaction algorithm will use them as heuristics in order to solve the problem.

The tool is not an automatic design configurer but a configuration *support* tool. This means that interaction with the KBS designer is necessary. Therefore we have chosen a mixed initiative approach. With this we allow the user to interrupt the flow of control whenever he does not agree with the computed deductions. It gives him the opportunity to change design decisions whenever he thinks this is necessary. The tool has to be adaptive to these changes and reconfigure its own reasoning process. The tool must also be able to perform user requests whenever the user so wants. In other words a user must be able to mediate between two extreme cases.

1. The user configures his design without making any use of the reasoning capabilities of the tool (i.e. only using the graphical interface as a support for visualisation).
2. The user relies completely on the reasoning aspect of the tool and only has to express approval for suggested solutions.

Functional requirements.

- Support a knowledge engineer in the design of a KBS.
- Make use of reusable design components
- Visualise and construct a design in terms of the *Common*Kads design model.
- Make explicit use of stated design knowledge (in terms of guidelines)
- Be able to explain the solution in terms of these guidelines together with the design decisions the user has made.

Decomposition: A functional task decomposition of the tool.

The top task "configuration design" is the task that the tool should support. It is decomposed into the following sub-tasks.

Collect analysis features. This task supports the acquisition of possible recognisable analysis features. They are generated out of the available guidelines and are stored in a library from which the user can select them.

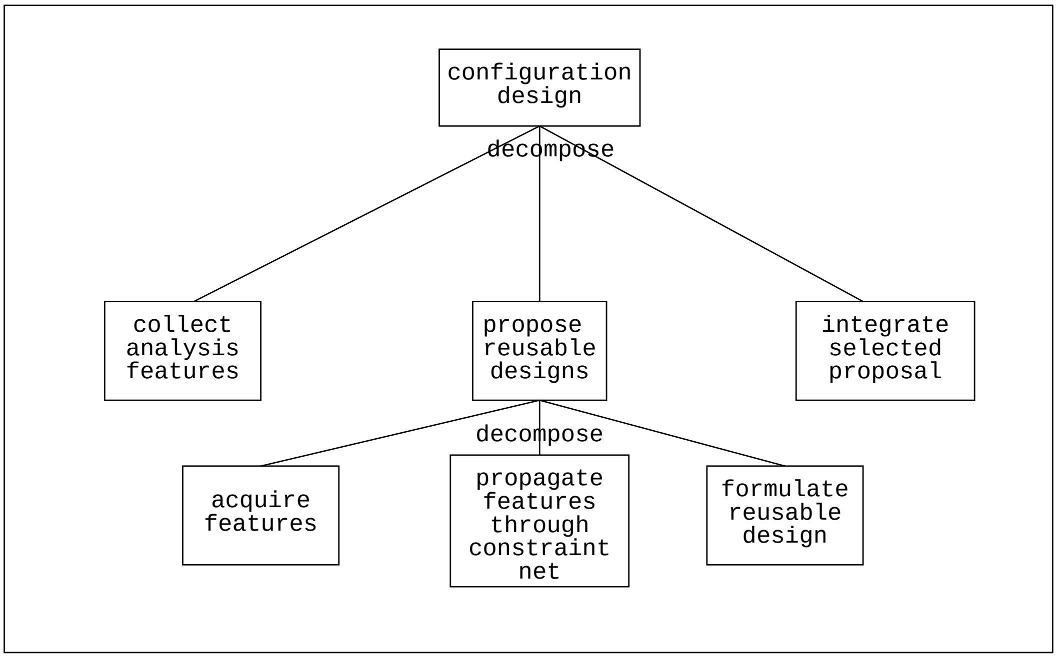


Figure 11.11: *Task decomposition of the configuration design tool.*

Propose reusable designs. This task has the goal to generate all possible reusable design component according to the given analysis features. It is further decomposed into the following three sub-tasks.

Acquire features. Select the given analysis features, put them in the right format and feed them to the constraint system.

Propagate features through constraint net. Activate the constraint satisfaction system.

Formulate reusable design. Retrieve a proposed design component from the constraint satisfaction system. Generate all the features that are attached to this component and put them in a format according to the *Common*Kads models.

Integrate selected proposal . Let the user select one of the proposed designs and integrate it in the existing design situation.

Architecture: Contax shell [Meyer, 1992] extended with X-windows interface calls.

Computational architecture: simple constraint satisfaction architecture.

Computational objects: guidelines formulated as constraints.

Computational method: forward constraint propagation.

Interface: Human-Computer

Interface paradigm: window based,

Interface medium: X-windows, motif style menus, buttons and popup windows.

Interface activities: mixed initiative, keyboard and mouse interactions.

Platform: Sun Spark workstation.

Target language: Harlequin lisp with integration of CLX (Common Lisp X interface) and CLUE (Common Lisp User Interface Environment).

11.5.3 System Interaction & Use

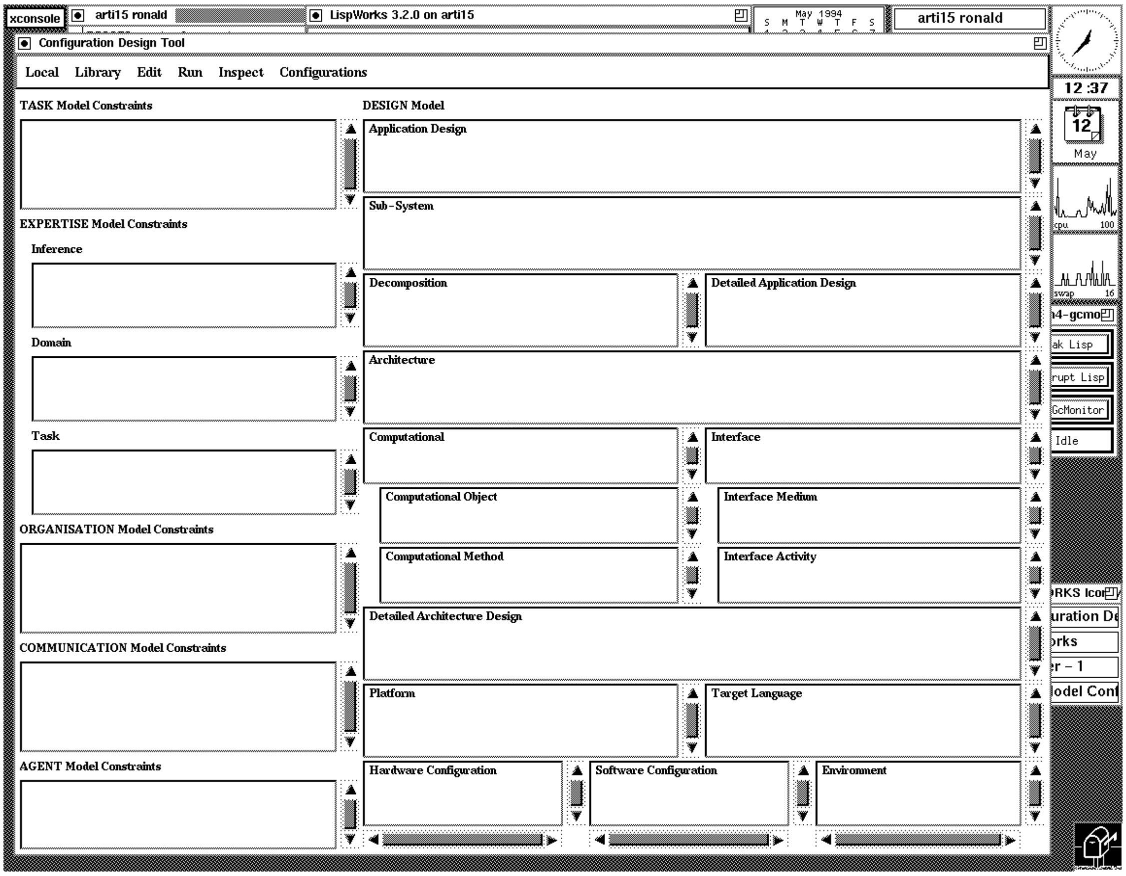


Figure 11.12: *Start up screen of the tool. On the left side are the KADS analysis models. The right side contains a detailed representation of the design model.*

Summary of the menu entries.

Library: The library menu gives the user the possibility to select recognisable analysis features which are related to a reusable design component via at least one design guideline. The features are extracted from the constraint net and provided to the user in a separate window for each type of analysis models (see Fig. 11.13).

Edit: Provides the necessary edit functionalities like inserting, deleting or changing instances of the tool.

Run: has two sub-menus.

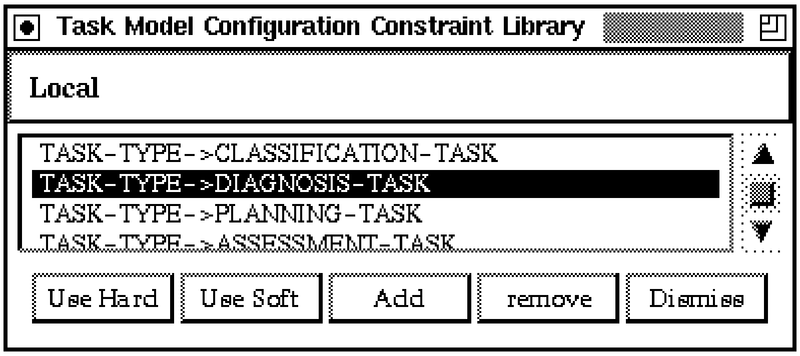


Figure 11.13: *Selection of Task analysis features. Select a feature and use it as a hard or soft constraint for the configuration of the design.*

- **Make Configuration:** Feeds all the filled-in constraints to the constraint propagation net and returns all the possible design configurations (see Fig. 11.14).
- **Reset Configuration:** Resets the constraint configuration system, i.e. the constraint propagation system will start from scratch the next time you call it.

Inspect: Includes four menu entries. The first three (**Domain Concepts, Constraints, Configuration**) activate tools that make it possible for the user to inspect the state of the underlying constraint satisfaction network. In the Domain Concepts window all the implemented domains are displayed (Fig. 11.15). The other two tools are harlequin lispworks inspectors with which you can browse through all the interesting configuration and constraint variables.

The fourth menu entry activates a monitoring window (Fig. 11.16). The configuration design tool monitors itself and reports all the meaningful actions that were taken by the user or the tool itself. This monitoring report can be made visible via the monitoring window. The monitor behaves like a text editor, and extra information can be added by the user and printed on paper or saved to a disk. Afterwards this report can serve as a rationale for the constructed design model.

Configurations: Contains a entry **Work Sheet** which gives the user the ability to go back to the state before configurations were generated (i.e. a kind of *undo*). If there are possible configurations available then a second entry becomes visible. This entry is called **Show Configurations** and provides a window from which the user can see all possible configurations and from which he can include one of these configurations into the work sheet at hand.

11.5.4 Evaluation of the tool

For the visualisation of the configuration design process we have chosen to represent all the *Common*Kads models in one framework. On first sight this may look confusing, but we found out that once you get used to this interface it provides a good overview of everything that happens during the configuration design process. Useful improvements to the tool include the ability to represent and inspect each model separately, and tool functionality to aid the user in entering or selecting requirements.

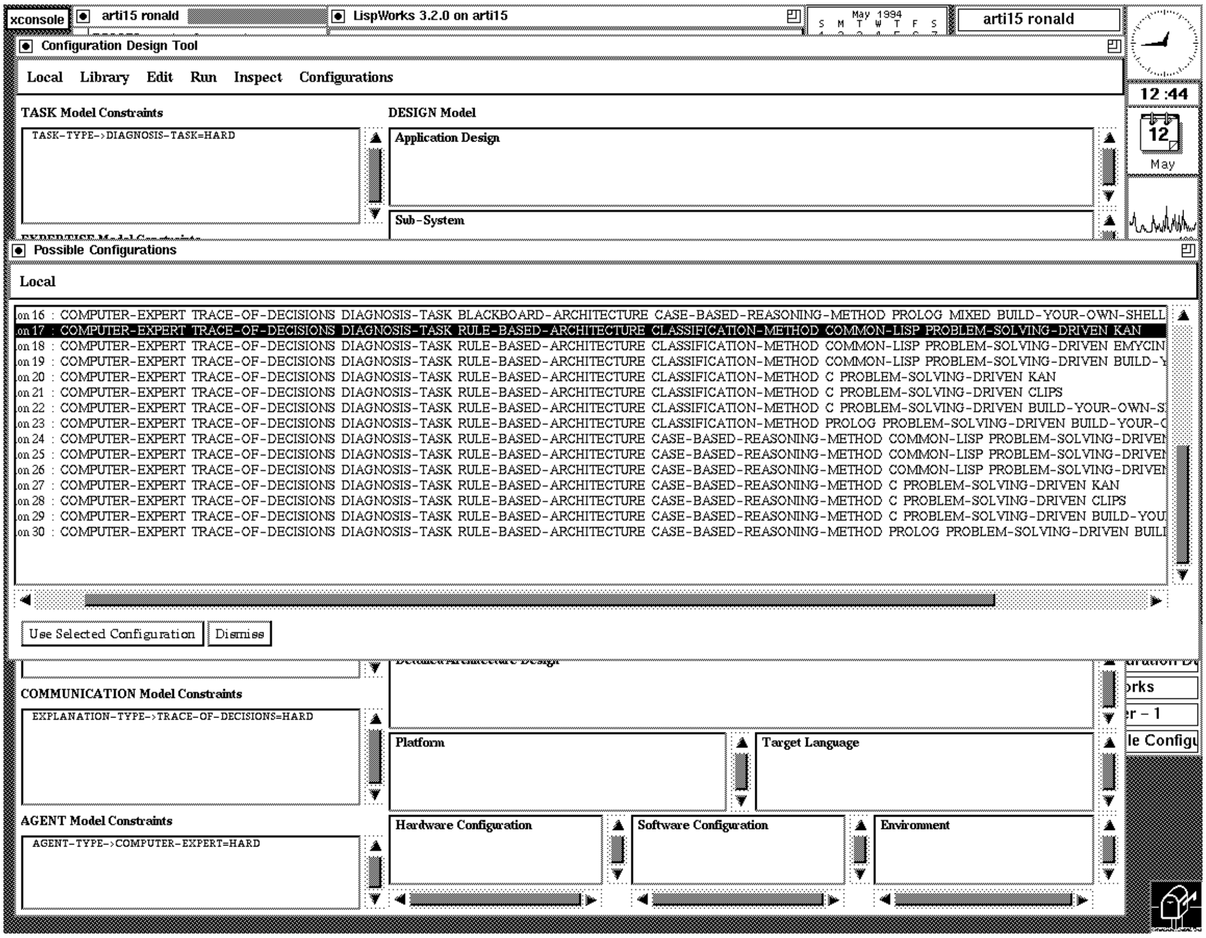


Figure 11.14: *The configuration tool after running a proposal for possible configurations.*

While testing we noticed that this environment could also be used to add new design configuration knowledge and new design components. At the moment this is only possible by writing the configurations yourself in the language that the Contax shell provides. In the successor of this tool it should be possible to add this kind of knowledge and extend the tool with new components via the simple graphical interface of the tool itself.

Explanation of why a certain component is selected by the configuration system is rudimentary at the moment. Instead of browsing through the internals of the configuration system or looking up some information in the monitoring window, we propose that the user should be able to retrieve this information in a much easier way. For example, there should be an explicit "Explain" button that provides a textual explanation of why a certain design component was chosen and how this choise influenced the rest of the *Common*Kads model. All this information can be retrieved from the system via the implemented design guidelines.

Selection of proposed configurations is not supported by the system. In order to give this support to a user, there should be worked out some kind of quality criteria for the selection and configuration of the design components, as well as for the integration of a configured design into the existing analysis situation. This quality metrics would make it

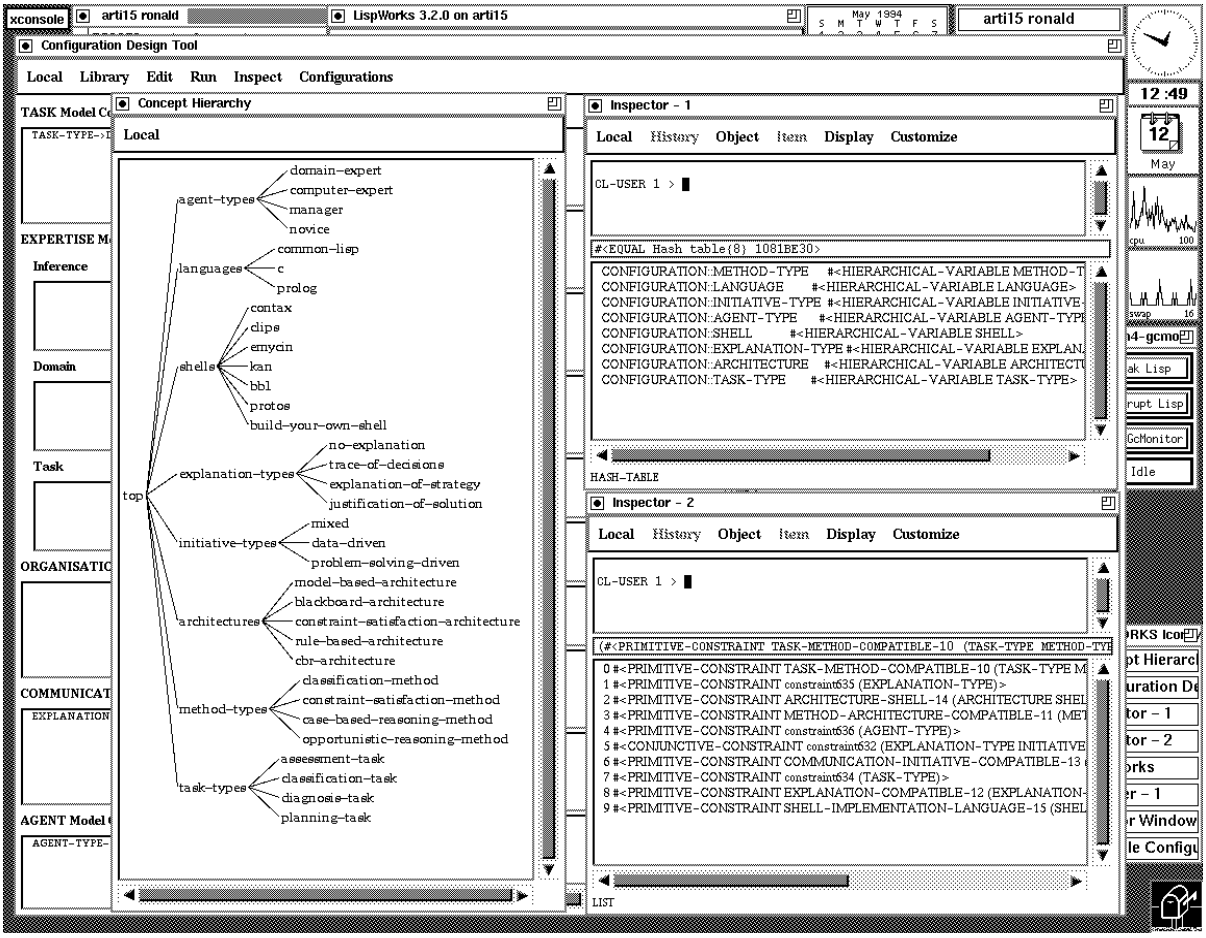


Figure 11.15: *Inspector tools.*

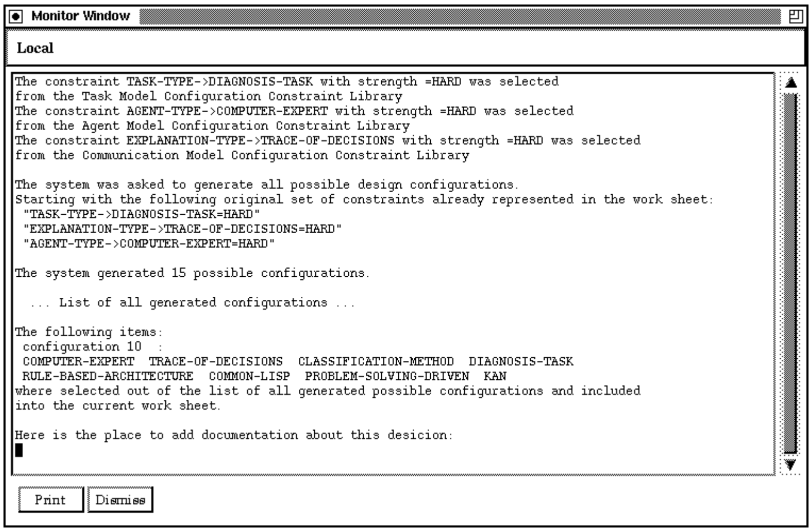


Figure 11.16: *The Monitoring window.*

possible for the system to propose a certain solution rather than some other ones.

Part III

Case Studies

Chapter 12

Sabena flight schedule case: an example of a design model

Ronald Schrooten

12.1 Introduction

The purpose of this chapter is to provide an example of a *Common*Kads design model. Although it does not evaluate the values and qualities of the design model itself, it can sure be helpful in the discussion about what should be in a design model and where and how should it be represented.
To achieve this goal we investigated a scheduling task in the crew scheduling department of Sabena. Previous analysis of this task was already done in the Task Model report [Duursma, 1993b]. For reasons of clarity we provide a brief summary of the task and communication model which have been developed in the Task Model report. Elaborating this task analysis resulted in the construction an expertise model and finally the required design model.

12.2 Task & Communication Model

This section contains a small extract of the Sabena flight schedule case **task model** and **communication model** which have been developed in [Duursma, 1993b]. We include it here to provide the reader with the necessary knowledge in order to understand the elaborated example for the rest of this text.

12.2.1 Task model

This part of the required task model describes the task at hand for the KBS to be built: propose a set of flight combinations given a set of individual afternoon flights each of which is associated with a set of morning flights.

task *propose flight combinations*

agent System

input Input set: A set of pairs of flight combinations

output Proposal: Proposed flights combinations

goal The Proposal should create pairs of a first and second element, where each pair contains at least one element, all pairs together cover all the elements in the input, and there is no output possible with more double pairs (i.e. output is an optimal combination).

12.2.2 Communication model

Part of the required communication model:

task *propose flight combinations*

input pairs: possible flight combinations
source: task selection pairs
initiative: task select pairs
format: standard pair format (to be defined)

output selection: A set of flight pairs
destination:
initiative: task select pairs
format: possible pair format (to be defined)

initiative User

user-role Commander

agent System

12.3 Expertise Model

A description of the *propose flight combinations* task in the Sabena flight schedule case example.

12.3.1 Domain knowledge:

domain ontology.

concept *flight*;
"a flight has one identifier of the type id, one departure time of the type time, and one arrival time of the type time."
properties

identifier: string;

departure: {00.00,00.01, ... ,23.59,24.00} cardinality: min 1; max 1;

arrival: {00.00,00.01, ... ,23.59,24.00} cardinality: min 1; max 1;

concept *morning-flight*;
sub-type-of: flight
property-values:

identifier: string;

departure: {00.00,00.01, ... ,11.59,12.00}

arrival: {00.00,00.01, ... ,11.59,12.00}

concept *afternoon-flight*;

sub-type-of: flight

property-values:

identifier: string;

departure: {12.00,12.01, ... ,23.59,24.00}

arrival: {12.00,12.01, ... ,23.59,24.00}

concept *flight-pair*;

"a flight pair consists of two flights which are ordered with respect to the relationes first-flight and second-flight"

properties

first-flight: morning-flight cardinality: min 0; max 1;

second-flight: afternoon-flight cardinality: min 0; max 1;

relation *can-be-combined*;

"a morning flight can be combine with an afternoon flight."

argument-1: morning-flight or first-flight;

argument-role: the-first-flight;

argument-2: afternoon-flight or second-flight;

argument-role: the-second-flight;

relation *selection-set*;

"a set of associations between a set of morning flights and an afternoon flight."

arguments:

all-morning-flights;

argument-role: all-possible-combinable-morning-flights
set of set(morning-flight);

cardinality: min 0; max infinite;

all-afternoon-flights;

argument-role: all-possible-combinable-afternoon-flight set of afternoon-flight;

cardinality: min 0; max infinite;

relation *associated*;

"a set of morning flights can be associated with an afternoon flight."

sub-type-of: selection-set;

argument-1: set(morning-flight)

argument-role: set-of-combinable-morning-flights;

cardinality: min 1; max 1;

differentiation-of: all-morning-flights(selection-set);

argument-2: afternoon-flight

argument-role: combinable-afternoon-flight;

cardinality: min 1; max 1;

differentiation-of: all-afternoon-flights(selection-set);

domain model

domain model *selection-set*;

parts: **template:** tuple(selection-set)

condition1: tuple(associated)

condition2: tuple(can-be-combined)

concept1: morning-flight

concept2: afternoon-flight

axioms: ∀ set(morning-flight) ∈ all-morning-flights
SELECTION-SET
∃ afternoon-flight ∈ all-afternoon-flights
SUCH THAT
set(morning-flights)
ASSOCIATED
afternoon-flight
AND
∀ morning-flight ∈ set(morning-flights)
CAN-BE-COMBINED
afternoon-flight

domain model *afternoon-flight-associations*;

parts: input1: selection-set;

input2: morning-flight;

condition: tuple(associated)

axioms: ∀ set(morning-flight) ∈ selection-set
SUCH THAT
morning-flight ∈ set(morning-flights)
AND
set(morning-flights)
ASSOCIATED
afternoon-flight

domain model *best-afternoon-flight*;

parts: constraint: morning-flight

input: tuple(selection-set)

condition: tuple(associated)

axioms: set(morning-flight) ∈ all-morning-flights
SUCH THAT
cardinality of morning-flights is minimal
AND
morning-flight ∈ set(morning-flights)
AND
set(morning-flights)
ASSOCIATED
afternoon-flight

annotations:

domain model *flight-pair*;

parts: input1: morning-flight;

input2: afternoon-flight;

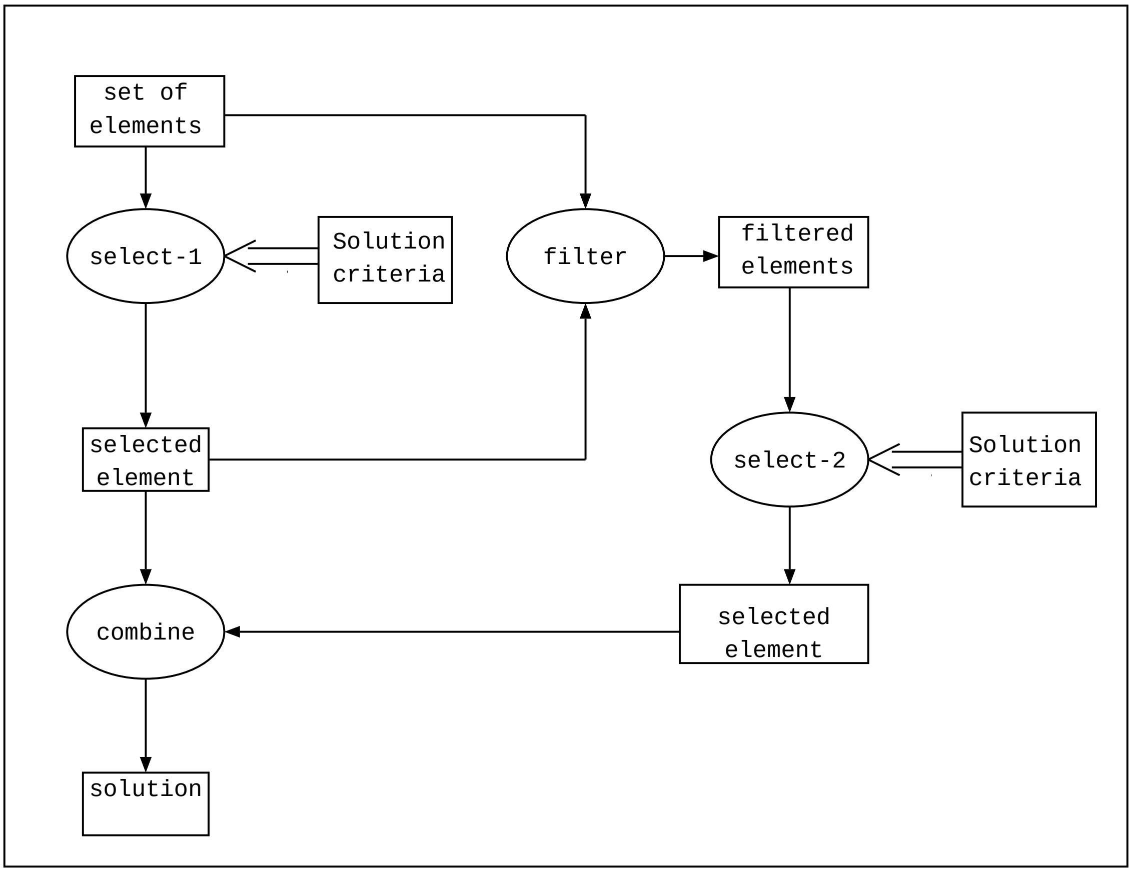
template: flight-pair

condition: tuple(can-be-combined)

axioms: morning-flight is first-flight of flight-pair
CAN-BE-COMBINED
afternoon-flight is second-flight of flight-pair

12.3.2 Inference knowledge:

Inference structure



Inference structure for the propose flight combinations task in the Sabena flight schedule case example.

Inference specification

inference *select-1;*
input-roles: *set of elements* → selection set
output-roles: *selected element* → morning flight
static-roles: *solution criteria* → morning flight ∈ selection-set

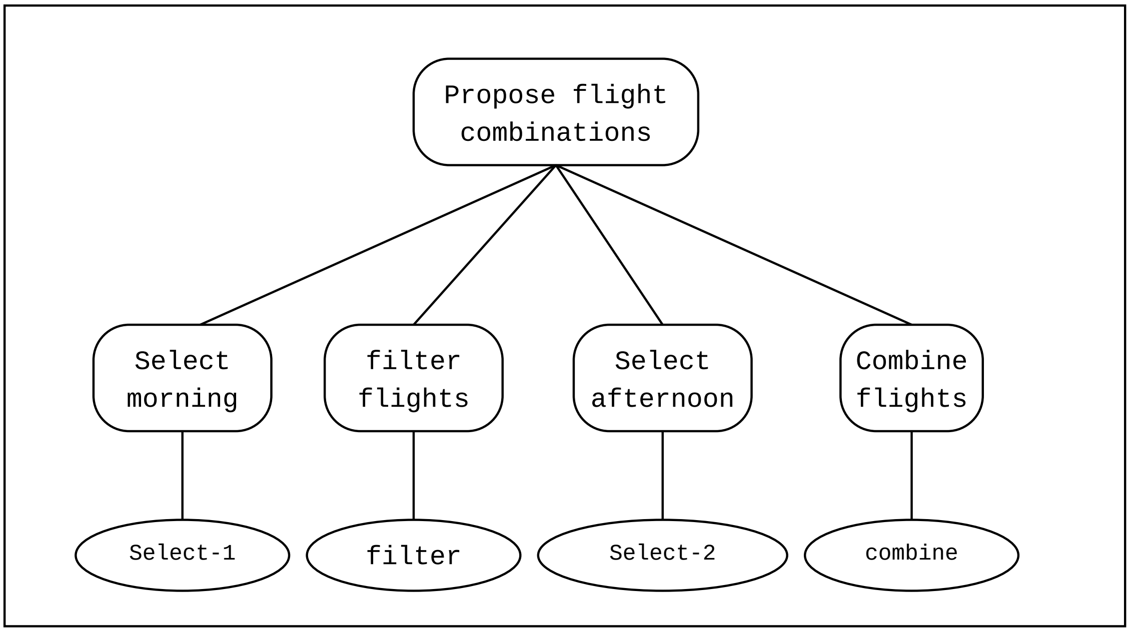
inference *select-2;*
input-roles: *filtered elements* → afternoon flight associations
output-roles: *selected element* → afternoon flight
static-roles: *solution criteria* → afternoon flight ∈ best afternoon flight criteria

inference *filter;*
input-roles: *set of elements* → selection set
selected element → morning flight
output-roles: *filtered elements* → afternoon flight associations

inference *combine;*
input-roles: *selected element* → afternoon flight
selected element → morning flight
output-roles: *solution* → flight pair

12.3.3 Task knowledge:

Task structure



Task structure of the task propose flight combinations in the task layer of the expertise model for the Sabena flight schedule case example.

Task knowledge

task *propose-flight-combinations;*
task-definition
goal:
set of flight combinations that satisfy the criteria;
input:
selection set: a set which contain afternoon flights respectively combined with all their possible morning flights.
criteria: conditions that should not be violated in the output set.
output:
proposal: a set of flight pairs that satisfy the goal.
spec: Flights provided by the input set are combined to flight pairs that meet the selection criteria.
task body:
type: composite;
sub-tasks:
select morning, filter flights, select afternoon, combine;
additional-roles:
morning flight: an element from the input set that satisfies the criteria of a morning flight

afternoon flight: an element from the input set that satisfies
the criteria of an afternoon flight and that also serves as the best
flight (according to the known criteria) that can be combined with a
certain morning flight.

control-structure:

propose-flight-combinations(+ selection-set, - proposal) =
Repeat
select-morning(+ selection-set, + solution-criteria, - morning-flight)
filter-flights(+ selection-set, + morning-flight, - filtered-elements)
select-afternoon(+ filtered-elements, + criteria, - afternoon-flight)
combine(+ morning-flight, + afternoon-flight, - flight-pair)
Until each element of the selection set is used

task *select morning;*

task-definition

goal:

find a morning flight that satisfy the criteria.

input:

selection set: a set which contain afternoon flights respectively
combined with all their possible morning flights.
solution criteria: knowledge about the element that should be
selected. Select an element that fits in the morning flight structure.

output:

morning flight: flight that satisfy the morning flight constraints.

spec: select one element of the input set according to
the selection criteria.

task body:

type: primitive;

sub-tasks:

select-morning;

control-structure:

select-1(+ selection-set, - morning-flight)

task *filter flights;*

task-definition

goal:

a set of flight associations which all have a certain morning flight in common.

input:

selection set: a set which contain afternoon flights respectively
combined with all their possible morning flights.
morning flight: flight that satisfies the morning flight constraints

output:

filtered elements: a set which contain afternoon flights respectively
combined with all their possible morning flights.

spec: take away all elements from a set that do not
have a certain characteristic

task body:

type: primitive;

sub-tasks:

filter;

control-structure:

filter(+ selection-set, + morning-flight, - filtered-elements)

task *select afternoon;*
task-definition
goal:
find an afternoon flight that satisfy the criteria.
input:
filtered elements: a set which contain afternoon flights respectively
combined with all their possible morning flights.
best afternoon flight criteria: knowledge about how to select
an afternoon flight from the input set.
output:
afternoon flight: flight that satisfy the afternoon flight
constraints and that seems to be the best flight that can be
combined with a certain morning flight.
spec: select one afternoon flight from the selection set
according to the criteria.
task body:
type: primitive;
sub-tasks:
select-afternoon;
control-structure:
select-2(+ filtered-elements,- afternoon-flight)

task *combine flights;*
task-definition
goal:
a flight pair
input:
morning flight: flight that satisfy the morning flight
constraints.
afternoon flight: flight that satisfy the afternoon flight
constraints.
output:
flight-pair: a combination of two elements. The first is an
morning flight or empty the second is an afternoon flight or
empty.
spec: combines a morning and an afternoon flight by putting
an order on the two input flights. If there is an input
missing then the remaining flight is combined with an empty
element.
task body:
type: primitive;
sub-tasks:
combine;
control-structure:
combine(+ morning-flight, +afternoon-flight, -flight-pair)

12.4 Design model

This construction is based on the design model as proposed in the this document.

For the textual representation of the design model we used the following notation convention: **name . extension** where name is a reference to the *name* of the model component and extension is a reference to the *type* of the component. We used abbreviations for the extensions, for example: **dapp** means that it belongs to the type "detailed application design" and **arc** is the extension name for the type "architecture".

12.4.1 High level design

Design model of proposed flight combinations.

propose-flight-combination.des: design model

consists-of: propose-flight-combination.app
set.arch
minimal-conventional-lisp.plat

Application design

propose-flight-combination.app: Application design

Application: propose-flight-combinations

application architecture: control over the sub-systems but since there is only one subsystem there is no top level control.

has-subsystems: propose-flight-combinations.sub

propose-flight-combinations.sub: Sub-system;

name: propose-flight-combinations;

type: KBS;

Functionality: "Proposes a set of flight combinations. It takes the domain model *selection-set* as an input set and filters out two elements using the criteria that are defined in the domain models *afternoon-flight-associations* and *best-afternoon-flight* and produces a set of elements that satisfy the conditions of the domain model *flight-pair*."

Implements: "The task *propose flight combinations* in the task model."

Has-decomposition: propose-flight-combinations.dec

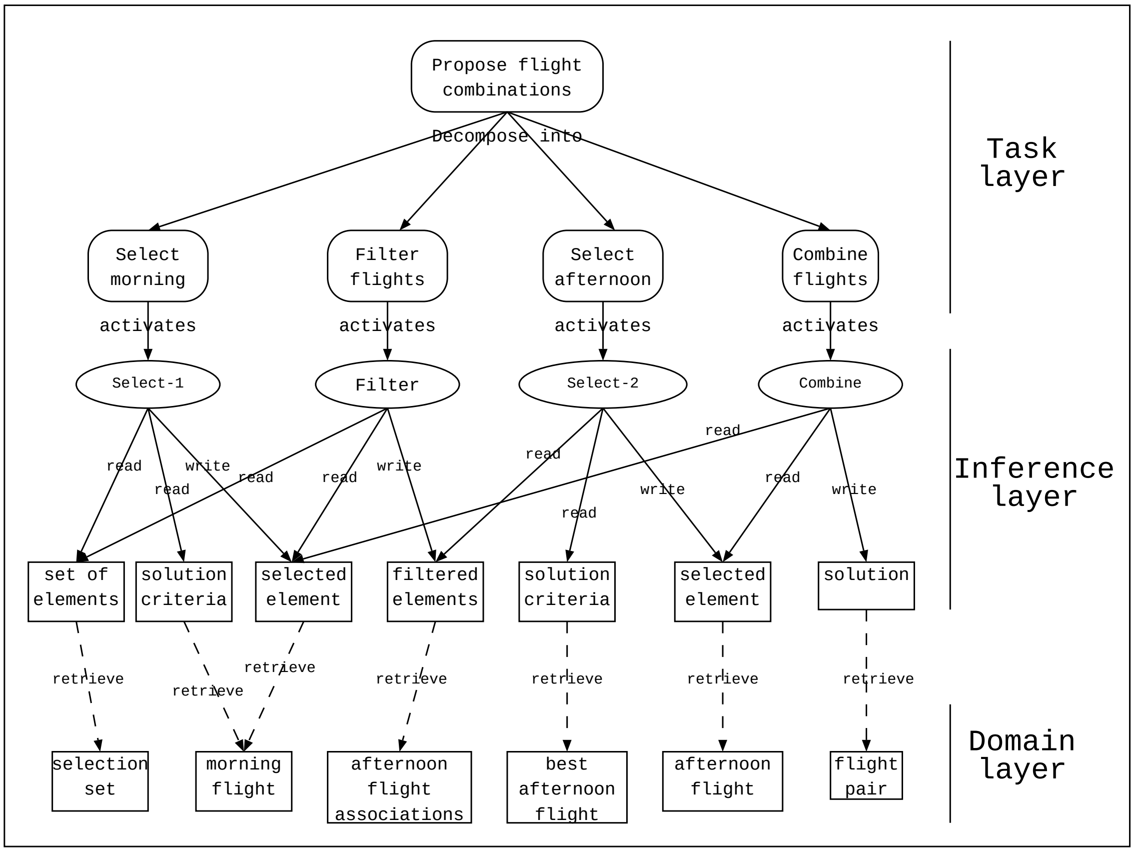
Has-detailed-design: propose-flight-combinations.dapp

propose-flight-combinations.dec: Decomposition

paradigm: Expertise;

DDL: HADL

program description: The decomposition of the subsystem into conceptual entities that realise the functionality.
From the task layer:
propose-flight-combinations; select-morning; filter-flights; select-afternoon; combine;
From the inference layer:
select-1; filter; select-2; combine;
from the domain layer:
domain ontology:
morning-flight; afternoon-flight; flight-pair; domain model:
selection-set; best-afternoon-flight; flight-pair;



Specification of the application subsystem decomposition.

12.4.1.1 Task layer

task module *propose-flight-combinations*
  refinement of propose-flight-combinations in model of expertise
  input
    Selection-set: combination set;
    Criteria: criteria set;
  output
    Proposal: pair set;
  control-terms
    all-done: controller;
    El1, El2: structure;
    Partial-set: combination set;
    Pair: pair;
  signature
    programs
      select-morning: combination set x structure;
      filter-flights: combination set x structure x combination set;
      select-afternoon: combination set x structure;
      combine-flights: structure x structure x pair;
  task-structures
    propose-flight-combinations (Selection-set, Proposal) $\equiv$
      WHILE ($\neg$ all-done(Selection-set)) DO
        select-morning(Selection-set, El1);
        filter-flights(Selection-set, El1, Partial-set);

```
            select-afternoon(Partial-set, EL2);
            combine-flights(El1, El2, Pair);
            add(Pair, Proposal)
          OD
end-task-module

task module select-morning
    refinement of select-morning in model of expertise
    input
      Selection-set: combination set;
      Criteria: criteria set;
    output
      EL: structure
    signature
      programs
        select: combination set x structure;
    task-structures
      select-morning(Selection-set, EL) ≡
      select (Selection-set, El)
end-task-module

task module filter-flights
    refinement of filter-flights in model of expertise
    input
      Selection-set: combination set;
      EL: structure;
    output
      Partial-set: combination set;
    signature
      programs
        filter: combination set x structure x combination set;
    task-structures
      filter-flights(Selection-set, EL, Partial-set) ≡
      filter (Selection-set, El, Partial-set)
end-task-module

task module select-afternoon
    refinement of select-afternoon in model of expertise
    input
      Partial-set: combination set;
      Criteria: criteria set;
    output
      EL: structure
    signature
      programs
        select: combination set x structure;
    task-structures
      select-afternoon(Selection-set, EL) ≡
      select (Selection-set, El)
end-task-module

task module combine-flights
    refinement of combine-flights in model of expertise
    input
      EL1: structure;
```

EL2: structure;
output
Pair: pair;
signature
programs
combine: structure x structure x pair;
task-structures
combine-flights(El1, EL2, Pair) ≡
combine(El1, EL2, Pair)
end-task-module

12.4.1.2 Inference layer

inference procedure *select-1*
operationalises select-morning
input from domain layer
set of elements
type set of pairs(set of atoms, atoms)
role type dynamic **from** selection set
mapping set of elements mapping **formalism** retrieve information
direction up **exhaustiveness** all
solution criteria
type function atom → boolean
role type static **from** morning flight
mapping solution criteria mapping **formalism** retrieve information
direction up **exhaustiveness** all
parameter output
selected element type atom
layer inference layer
method random: set → element
method find-all: set x key → set
spec selected-element := random(find-all(selection-set, selection-criteria))
end procedure

inference procedure *filter*
operationalises filter flights
input from domain layer
set of elements
type set of pairs(set of atoms, atoms)
role type dynamic **from** selection set
mapping set of elements mapping **formalism** retrieve information
direction up **exhaustiveness** all
parameter input
selected element **type** atom
parameter output
filtered elements **type** set of pairs(set of atom, atom)
layer inference layer
local variables
element **type** set of atom
subset **type** pairs(element, atom)
method push-el: element set → set
method member: atom x set → boolean
spec
for-all element in set-of-elements do
if member(selected-element, element)
then

filtered-elements:= push-el(element,filtered-elements)
end procedure

inference procedure *select-2*
operationalises select-afternoon
parameter input
filtered elements **type** set of pairs(set of atom, atom)
input from domain layer
solution criteria
type function set → boolean
role type static **from** best afternoon flight
mapping solution criteria mapping **formalism** retrieve information
direction up **exhaustiveness** all
parameter output
selected element type atom
layer inference layer
method smallest: set → element
method find-all: set x key → set
spec
selected-element := smallest(find-all(selection-set,selection-criteria))
end procedure

inference procedure *combine*
operationalises combine-flights
parameter input
selected-element-1 type atom
selected-element-2 type atom
output to domain layer
solution
type pairs(atom,atom)
role type dynamic **from** flight-pair
mapping solution mapping **formalism** provide information
direction down **exhaustiveness** all
layer inference layer
method add-to-structure: structure atom → structure
spec
solution := add-to-structure(solution :first-el selected-element-1 :second-el selected-element-2)
end procedure

12.4.1.3 Domain layer

The structure and functionality of the domain layer concepts is the same as described in the model of expertise.

12.4.2 Detailed design

select-afternoon.dapp: Detailed application design

PDL: BNF of the program description language.

```
program-description    ::= program-line +
program-line           ::= < architecture-command ( program-line * ) > * |
                           < architecture-command ( object-description * ) ; > * |
                           control-block *
object-description     ::= < object-type instance > +
object-type            ::= : "computational object in the architecture"
```

instance	::= "string"
architecture-command	::= "command in the architecture"
control-block	::= architecture-command program-line architecture-command

program description :

repeat
get-el(:special-set selection-set :el first-element :key morning-flight);
sub-set(:special-set selection-set :special-set afternoon-flight-associations :key first-flight);
get-el(:special-set afternoon-flight-associations :el second-element :key best-afternoon-flight);
make-el(:el-name flight-pair :characteristic-1 first-flight :characteristic-2 second-flight)
remove-el(:element first-flight :special-set selection-set);
remove-el(:element second-flight :special-set selection-set);
until empty(:special-set selection-set);

Realised-within: set.arch;

Instantiates: set.obj;

Calls: set.meth;

12.4.3 Architecture design

set.arch: Architecture design

computational: set.comp;

set.comp: Computational

paradigm: set manipulation;

command: repeat, until, get-el, subset, make-el, remove-el

uses: set.obj;

supports: set.meth;

has-detailed-design: setc.dar;

set.obj: Computational object

set: "a set that covers several elements"

element: "a data object that can hold specific characteristics"

special-structure: "a structure which has two slots, the first slot contains a set and the second an element."

special-set: "a set of special structures"

key: "holds criteria that can be tested"

set.meth: Computational method

internal-control: decomposes into
start
read
apply
stop

explicit-control: consists of
repeat *internal-control* **until** *stop-condition*

get-el: "get an element of the type element from the special-set according to the criteria in the key slot."

subset: "get a subset of the original input set and each element in this subset contains the specified key element"

make-el: "make an element with the specified characteristics"

remove-el: "remove all elements that match with a given element out of the special-set"

empty: "test if a set is empty"

Works-on :

get-el
WORKS-ON
special-set, el, key;
subset
WORKS-ON
special-set, key;
make-el
WORKS-ON
element;
remove-el
WORKS-ON
el, special-set;
empty
WORKS-ON
special-set;

set.darc: Detailed architecture design

PDL: The program description language will be according to the BNF of Lisp.

program description: "Specification of the computational methods and objects that are mentioned in the architecture. This specification will be in terms of lisp."

realised-within: lisp.targ;

12.4.4 Platform design

conventional.plat: Platform design

Uses Language: lisp.targ;

Required hardware: conventional.hard;

Required software: lisp.soft;

Has user: user in agent model;

Has system model: model of the system in agent model;

User-environment: .env;

lisp.targ: Target language: LISP

.user: User: revise agent "(Agent model (task and communication model))"

conventional.hard: Hardware configuration: "minimal conventional computer configuration, but pay attention that the available RAM is not under 5 Mbyte."

available in: computing resource in the organisational model

lisp.soft: Software configuration: Lisp package

available in: other resource in the organisational model

.env: Environment: according to hardware configuration

available in: other resource in the organisational model

Bibliography

[9001, 1987] 9001, I. (1987). Quality systems - model for quality assurance in design/development production and servicing.

[Aben, 1993] Aben, M., Balder, J., van Harmelen, F. (1993). Support for the formalization and validation of KADS expertise models KADS-II/M2/UvA/DM2.6a/1.0, Social Science Informatics, University of Amsterdam.

[Aiello, 1986] Aiello, N. (1986). User-directed of parallelism: The cage system,. *KSL 86-31, Stanford University, Stanford, CA.*

[Angele *et al.*, 1991] Angele, J., Fensel, D., Landes, D., & Studer, R. (1991). KARL: An executable language for the conceptual model. In *Proceedings of the 6th Banff Knowledge Acquisition for Knowledge-Based Systems Workshop*, pages 1.1–20, Banff, Canada. SRDG Publications, University of Calgary.

[Bahrami, 1988] Bahrami, A. (1988). *Designing Artificial Intelligence Based Software.* Sigma Press, South Oak Lane, Wilslow, SK9 6AR, England.

[Balder *et al.*, 1992] Balder, J., van Harmelen, F., & Aben, M. (1992). A KADS/ML^2 model of a scheduling task. In van Langeveld, I. & Treur, J., editors, *Proceedings of the international workshop on formal specification methods for complex reasoning systems*, pages 15–42, Vienna. (at ECAI'92).

[Balzer and Williams, 1980] Balzer, R.; Erman, L. L. P. and Williams, C. (1980). Hearsay-iii: A domain-independent framework for expert systems. In *Proc. 1980 AAAI Conf.*, Stanford, CA.

[Bernaras, 1993] Bernaras, A. (1993). Models of Design for the CommonKADS Library. KADS-II/M2.3/VUB/DD//1.0, VUB.

[Benjamins *et al.*, 1992] Benjamins, V. R., Jansweijer, W. N. H., & Abu-Hanna, A. (1992). Integrating problem solving methods into KADS. In Bauer, C. & Karbach, W., editors, *Proccedings 2nd KADS User Meeting*, Munich. SKBS/A2/92-02.

[Billault, 1989] Billault, J. P. (1989). Design and implementation of a configuration task. ESPRIT Project P1098, Deliverable E5.2, University of Amsterdam. Available from: University of Amsterdam, Social Science Informatics, Roetersstraat 15, 1018 WB, The Netherlands.

[Breuker *et al.*, 1987] 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.

[Breuker and Van de Velde, 1993] Breuker, J. A. and Van de Velde, W. (1993). The commonkads library. Technical Report ESPRIT Project P5248 KADS-II/M2/VUB/TR/054/1.1, The KADS-II Consortium.

[Bylander & Mittal, 1986] Bylander, T. & Mittal, S. (1986). CSRL: A language for classificatory problem solving and uncertainty handling. *AI Magazine*, 7.

[Chandrasekaran, 1988] 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, 1990] Chandrasekaran, B. (1990). Design problem solving: A task analysis. *AI Magazine*, 11(4), 59–71.

[Clancey, 1985] Clancey, W. J. (1985). Acquiring, representing and evaluating a competence model of diagnostic strategy. In Chi, Glaser, & Far, editors, *Contributions to the Nature of Expertise.*

[Clancey, 1985] Clancey, W. J. (1985). Heuristic classification. *Artificial Intelligence*, 27, 289–350.

[Clancey and Letsinger, 1984] Clancey, W. J. and Letsinger, R. (1984) NEOMYCIN: Reconfiguring a rulebased expert system for application to teaching. In Clancey, W. J. and Shortliffe, E. H. (Eds.), *Readings in Medical Artificial Intelligence: the first decade*, 361-381. Addison-Wesley, Reading.

[Coad and Yourdon, 1991] Coad, P. and Yourdon, E. (1991). *Object-Oriented Analysis.* Prentice Hall, Englewood Cliffs, New Jersey.

[Corkill and Murray, 1986] Corkill, D.D., G. K. and Murray, K. (1986). Gbb: A generic blackboard development system. In *Proc. AAAI-86*, 1008–1014, Philadelphia, PA. AAAI.

[David and Krivine, 1990] David, J. M. and Krivine, J. P. (1990) Explaining reasoning from knowledge level models In Aiello, L. (Ed), *Proceedings ECAI'90, Stockholm*, 186-188, London, Pitman.

[de Greef, 1989] de Greef, P. (1989). Control of cooperation. the design of a knowledge based system that can operate in different modalities. Technical report, University of Amsterdam.

[de Hoog et al., 1993b] de Hoog, R., Martil, R., Wielinga, B., Taylor, R., Bright, C., and Van de Velde, W. (1993b). The CommonKADS model set. ESPRIT Project P5248 KADS-II/M1/DM..1b/UvA/018/5.0, University of Amsterdam, Lloyd's Register, Touche Ross Management Consultants & Free University of Brussels.

[de Hoog et al., 1993] DE HOOG, R., BENUS, B., METSELAAR, C., AND VOGLER, M. (1993). The common KADS organisation model. ESPRIT Project P5248 KADS II KADS-II/TM6/DM6.2/UvA/041/1.0, Social Science Informatics, University of Amsterdam.

[Devanbu et al., 1991] DEVANBU, P., BRACHMAN, R.J., SELFRIDGE, P.G., AND BALLARD, B. (1991) A knowledge-based software information system *Communications of the ACM*, 1991, May, vol 34, no 5, 35-49

[Duursma, 1992a] DUURSMA, C. (1992a). Interpretation models and problem solving methods. Vrije Universiteit Brussel, AI-Lab.

[Duursma, 1992b] DUURSMA, C. (1992b). Role limiting methods for the KADS-II conceptual model library. ESPRIT Project P5248 AI-Memo 92/4, AI-Lab, Vrije Universiteit Brussel, Pleinlaan 2, 1050 Brussels, Belgium.

[Duursma, 1993a] DUURSMA, C. (1993). Expertise assessment through theory evaluation. Technical Report AI-Memo 93/7, AI-Lab, Vrije Universiteit Brussel, Brussels, Belgium.

[Duursma, 1993b] DUURSMA, C. (1993). Task model definition and task analysis process. Deliverable M.5, Version 1.1b ESPRIT Project P524 KADS-II/M5/VUB/RR/004/1.1b, The KADS-II Consortium.

[Duursma et al., 1992a] DUURSMA, C., AAMODT, A., AND VAN DE VELDE, W. (1992a). Requirements definition, task analysis and specifications of the CM-editor. Esprit project p5248, AI-Lab, Vrije Universiteit Brussel, Pleinlaan 2 Brussels.

[Duursma et al., 1993] DUURSMA, C., CAÑAMERO, D., AND SCHROOTEN, R. (1993). Model ontology requirements for domain model validation. In Meseguer, P. and Cardenosa, J. (Eds.). , *Proceedings of EUROVAV 93.*

[Duursma et al., 1992] DUURSMA, C., SCHREIBER, G., TERPSTRA, P., AND DE VELDE, W. V. (1992). Operationalising knowledge models. ESPRIT Project P5248 KADS-II, Deliverable D1.2.3 KADS-II/T1.2/VUB/TR/004/1.0, UvA, VUB.

[Duursma *et al.*, 1992] DUURSMA, C., VANDEVELDE, W., SCHREIBER, G., & TERPSTRA, P. (1992). Operationalising knowledge models. Deliverable KADS-II/T1.2/VUB/RR/004/1.0, VuB.

[de Kleer, 1986] DE KLEER, J. (1986). An assumption-based TMS. *Artificial Intelligence*, 28:127–162.

[Engelmore and Terry, 1979] ENGELMORE, R. S. AND TERRY, A. (1979). Structure and function of the crysalis system. In *Proc. 6th IJCAI*, 250–256, Tokyo, Japan. IJCAI.

[Erman and Reddy, 1980] ERMAN, L.D., H.-R. F. L. V. AND REDDY, D. (1980). The hearsay-ii speech-understanding system: Integrating knowledge to resolve uncertainty. *Computing Surveys*, 12, 213–253.

[Eshelman, 1988b] ESHELMAN, L. (1988). MOLE: A knowledge-acquisition tool for cover-and-differentiate systems. In Marcus, S. (Ed.), *Automating Knowledge Acquisition for Expert Systems*, 37-80. Kluwer Academic Publishers, The Netherlands.

[Fenton, 1992] FENTON, N. E. (1992). *Software metrics. A regiolous approach (First Edition)*. Chapman and Hall, London.

[Forte and McCulley, 1991] FORTE, G. AND MCCULLEY, K. (EDS.). (1991). *CASE Outlook: Guide to Products and Services*. CASE Consulting Group, Lake Oswego, Ore.

[Fuggetta, 1993] FUGGETTA, A. (1993). A classification of case technology. *Computer*, 26(12), 25–38.

[Gaines, 1994] GAINES, B. (ED.). (1994). *International Journal of Human - Computer Studies*, volume 40, number 2. Academic Press.

[Goel *et al.*, 1987] GOEL, A., SOUNDARAJAN, N., & CHANDRASEKARAN, B. (1987). Complexity in classificatory reasoning. In *AAAI–87*, pages 421–425.

[Golfinopoulos, 1993] GOLFINOPOULOS, V. (1993). Quality assurance for blackboard systems. Master thesis. Vrije Universiteit Brussel, Faculty of Applied Sciences.

[de Greef *et al.*, 1988] DE GREEF, P., BREUKER, J. A., SCHREIBER, A. T., & WIELEMAKER, J. (1988). StatCons: Knowledge acquisition in a complex domain. In *Proceedings ECAI–88*, Munich.

[de Greef *et al.*, 1987] DE GREEF, P., SCHREIBER, A. T., & WIELEMAKER, J. (1987). The StatCons case study. ESPRIT Project P1098, Deliverable E2.3 (experiment F2), University of Amsterdam. Available from: University of Amsterdam, Social Science Informatics, Roetersstraat 15, 1018 WB, The Netherlands.

[Green et al., 1986] GREEN, C., LUCKHAM, D., BALZER, R., CHEATHAM, T., AND RICH, C. (1986). Report on a knowledge-based software assistant. In Rich, C. and Waters, R. C. (Eds.). , *Readings in Artificial Intelligence and Software Engineering*, 377–428. Morgan Kaufmann, San Mateo, CA.

[Hayes-Roth, 1985] HAYES-ROTH, B. (1985). A blackboard architecture for control. *Artificial Intelligence*, 26(3), 251–321.

[Hemman, 1992] HEMMAN, T. (1992) How knowledge engineering can benefit from software engineering with respect to reuse: Towards reusable knowledge models In Constantopoulos, P. and Vassiliou, Y. (Eds), *Methods and Tools for Software Reuse*, 212-227. ERCIM Workshop Report ERCIM-92-W006, 1992

[ISO9126, 1990] ISO9126 (1990). Iso/iec 9126 information technology - software product evaluation - quality characteristics and guidelines fotr their use.

[Jansweijer *et al.*, 1986] JANSWEIJER, W. N. H., ELSHOUT, J. J., & WIELINGA, B. J. (1986). The expertise of novice problem solvers. In *Proceedings ECAI–86, Brigthon*.

[Jonker & Spee, 1992] JONKER, W. & SPEE, J. W. (1992). Yet another formalisation of KADS conceptual models. In Wetter, T., editor, *Proceedings of European Knowledge Acquisition Workshop EKAW'92.*

[Karbach *et al.*, 1988] KARBACH, W., TONG, X., & VOSS, A. (1988). Filling in the knowledge acquisition gap: via KADS models of expertise to ZDEST–2 expert systems. In *Proceedings of EKAW 88*, Bonn.

[Karbach *et al.*, 1991] KARBACH, W., VOSS, A., SCHUKEY, R., & DROUWEN, U. (1991). Model-K: Prototyping at the knowledge level. In *Proceedings Expert Systems–91, Avignon, France*, pages 501–512.

[Killin *et al.*, 1991] KILLIN, J., MORGAN-GRAY, E., & PORTER, D. (1991). Knowledge engineering within software engineering: Similarities and differences. In *Proceedings of the British Computer Society Specialist Group on Expert Systems 1st International Workshop on KBS Methodologies.*

[Killin, 1993] KILLIN, J. (1993). Managing and maintaining an operational KADS system models. In Schreiber, G., editor, *KADS book.*

[Kline and Dolins, 1989] KLINE, P. J. AND DOLINS, S. B. (1989). *Designing expert systems. A guide to selecting implementation techniques.* Wiley, New York.

[Klinker *et al.*, 1991] KLINKER, G., BHOLA, C., DALLEMAGNE, G., MARQUES, D., & MCDERMOTT, J. (1991). Usable and reusable programming constructs. *Knowledge Acquisition*, 3:117–136.

[Lemmers, 1991] LEMMERS, M. (1991). A shell for systematic diagnosis: Structure-preserving design of a KBS. Master's thesis, University of Amsterdam, Social Science Informatics.

[Linster, 1992] LINSTER, M. (1992). *Knowledge acquisition based on explicit methods of problem–solving.* PhD thesis, University of Kaiserslautern.

[Lowry, 1992] LOWRY, M. R. (1992). Software engineering in the twenty-first century. *AI Magazine*, Fall, 71–87.

[Mackworth, 1977] MACKWORTH, A. (1977). On reading sketch maps. In *Proceedings of IJCAI–77*, 598 – 606. International Joint Conference on Artificial Intelligence.

[Mackworth, 1987] MACKWORTH, A. (1987). Question answering. In Shapiro, S. C. (Ed.). , *Encyclopedia of Artificial Intelligence*, 205–211. John Wiley and Sons, New York.

[Maes, 1987] MAES, P. (1987). Computational reflection. Technical report 87-2, Free University of Brussels, AI Lab.

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

[McDermott, 1989] McDermott, J. (1989) Preliminary steps towards a taxonomy of problem-solving methods In Marcus, S., editor, *Automating Knowledge Acquisition for Expert Systems*, pages 225–255. Kluwer Academic Publisher, The Netherlands.

[Meyer, 1992] Meyer, M. (1992). Using hierarchical constraint satisfaction for lathe-tool selection in a cim environment. Technical Report RR 92-35, DFKI Gmbh, Postfach 20 80 D-6750 Kaiserslautern, Germany.

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

[Musen *et al.*, 1987] Musen, M. A., Fagan, L. M., Combs, D. M., & Shortliffe, E. H. (1987). Use of a domain-model to drive an interactive knowledge-editing tool. *International Journal of Man-Machine Studies*, 26:105–121.

[Neches et al., 1985] Neches, R., Swartout, W. R., and Moore, J. D. (1985). Enhanced maintenance and explanation of expert systems through explicit models of their development. *IEEE Transactions on Software Engineering*, 11, 1337-1351.

[Nii and Aiello, 1979] Nii, H. P. and Aiello, N. (1979). AGE (attempt to generallize): A knowledge-based program for building knowledge based programs. In *IJCAI–79*, 645–655.

[Nii, 1986] Nii, P. (1986). The blackboard model of problem solving. *AI Magazine*, VII(2), 38–53.

[Partridge, 1992] Partridge, D. (1992). *Engineering Artificial Intelligence Software.* Intellect, Oxford, England.

[Porter, 1992] Porter, D. (1992). Reusable analysis and design components for KBS development. In *Current Developments in Knowledge Acquisition: Proceedings of EKAW'92*, Heidelberg, Germany. Springer-Verlag.

[Prieto-Diaz and Freeman, 1987] Prieto-Diaz, R. and Freeman, P. (1987) Classifying software for reusability *IEEE Software*, 1987, January, vol 4, no 1

[Prieto-Diaz, 1991] Prieto-Diaz, R. (1991) Implementing faceted classification for software reuse *Communications of the ACM*, 1991, May, vol 34, no 5, 89-97

[Puerta *et al.*, 1991] Puerta, A., Egar, J., Tu, S. W., & Musen, M. A. (1991). A multiple-method knowledge-acquisition shell for the automatic generation of knowledge-acquisition tools. In *Proceedings of the 6th Knowledge Acquisition for Knowledge-Based Systems Workshop, Banff, Canada*, pages 20.1–19. SRDG Publications, University of Calgary.

[Rademakers, 1991] Rademakers, P. (1991). Task analysis of an equipment assignment problem. AI Memo 91-1, AI-Lab, Vrije Universiteit Brussel, Brussels, Belgium.

[Reichgelt & van Harmelen, 1986] Reichgelt, H. & van Harmelen, F. (1986). Criteria for choosing represenation languages and control regimes for expert systems. *Knowledge Engineering Review*, 1:2–17.

[Reinders *et al.*, 1991] REINDERS, M., VINKHUYZEN, E., VOSS, A., AKKERMANS, J. M., BALDER, J. R., BARTSCH-SPÖRL, B., BREDEWEG, B., DROUVEN, U., VAN HARMELEN, F., KARBACH, W., KARSSEN, Z., SCHREIBER, A. T., & WIELINGA, B. J. (1991). A conceptual modelling framework for knowledge-level reflection. *AI Communications*, 4(2-3):74–87.

[Rice, 1986] RICE, J. P. (1986). Poloigon, a system for parrallel problem solving. *KSL 86-19, Stanford University, Stanford, CA.*

[Rumbaugh et al., 1991] RUMBAUGH, J., BLAHA, M., PREMERLANI, W., EDDY, F., AND LORENSEN, W. (1991). *Object-Oriented Modelling and Design.* Prentice Hall, Englewood Cliffs, New Jersey.

[Schreiber, 1992] SCHREIBER, A. T. (1992). *Pragmatics of the Knowledge Level.* PhD thesis, University of Amsterdam.

[Schreiber, 1992] SCHREIBER, A. T. (1992). Applying KADS to the sisyphus domain. In Schreiber, A. T., Wielinga, B. J., and Breuker, J. A. (Eds.). , *KADS: A Principled Approach to Knowledge-Based System Development.* Academic Press, London. Forthcoming.

[Schreiber, 1992a] SCHREIBER, A. T. (1992a). A KADS domain description language. In Schreiber, A. T., Wielinga, B. J., & Breuker, J. A., editors, *KADS: A Principled Approach to Knowledge-Based System Development.* Academic Press, London. Forthcoming.

[Schreiber et al., 1987] SCHREIBER, A. T., BREDEWEG, B., DAVOODI, M., WIELINGA, B.J. (1987) Toward A Design Methodology For KBS ESPRIT Project P1098, Deliverable D8 Uva/Stc-B2-Pr–001, Vf Memo 97 University Of Amsterdam Available from: University of Amsterdam, Social Science Informatics, Roetersstraat 15, 1018 WB, The Netherlands.

[Schreiber et al., 1989] SCHREIBER, A. T., BREDEWEG, B., DE GREEF, P., TERPSTRA, P., WIELINGA, B. J., BRUNET, E., SIMONIN, N., AND WALLYN, A. (1989). A KADS approach to KBS design. ESPRIT Project 1098, deliverable B6 UvA-B6-PR–010, University of Amsterdam & Cap Sogeti Innovation. Available from: University of Amsterdam, Social Science Informatics, Roetersstraat 15, 1018 WB, The Netherlands.

[Schreiber et al., 1990] SCHREIBER, A. TH., WIELINGA, B.J., HESKETH, P., LEWIS, A. (1990) A KADS Design Description Language ESPRIT Project 1098, deliverable B7, University of Amsterdam & STC Technology Ltd. Available from: University of Amsterdam, Social Science Informatics, Roetersstraat 15, 1018 WB, The Netherlands.

[Schreiber *et al.*, 1991a] SCHREIBER, A. T., AKKERMANS, J. M., & WIELINGA, B. J. (1991a). On problems with the knowledge level perspective. In Steels, L. & Smith, B., editors, *AISB–91: Artificial Intelligence and Simulation of behaviour*, pages 208–221, London. Springer Verlag. Also in: *Proceedings Banff–90 Knowledge Acquisition Workshop*, J. H. Boose and B. R. Gaines (editors), SRDG Publications, University of Calgary, pages 30-1 – 30-14.

[Schrooten and Duursma, 1994] SCHROOTEN, R. AND DUURSMA, C. (1994). Configuration support environment functional specification. ESPRIT Project P5248 KADS II DM7.4/VUB/RR/004/1.0, AI-Lab, Vrije Universiteit Brussel, Brussels, Belgium.

[Schreiber *et al.*, 1991b] SCHREIBER, A. T., BARTSCH-SPÖRL, B., BREDEWEG, B., VAN HARMELEN, F., KARBACH, W., REINDERS, M., VINKHUYZEN, E., & VOSS, A. (1991b). Designing architectures for knowledge-level reflection. ESPRIT Basic Research Action P3178 REFLECT, Deliverable IR.4 RFL/UvA/III.1/4, REFLECT Consortium. Available from: University of Amsterdam, Social Science Informatics, Roetersstraat 15, 1018 WB, The Netherlands.

[Shadbolt and Wielinga, 1990] SHADBOLT, N. AND WIELINGA, B. J. (1990) Knowledge based knowledge acquisition: the next generation of support tools In Wielinga, B. J., Boose, J., Gaines, B., Schreiber, G., and van Someren, M. W. (Eds.) *Current trends in Knowledge Acquisition*, 313-338, Amsterdam, IOS Press.

[Smith, 1985] SMITH, B. C. (1985). Prologue to "reflection and semantics in a procedural language". In Brachman, R. J. & Levesque, H. J., editors, *Readings in Knowledge Representation*, pages 31–40. Morgan Kaufman, California.

[Smith, 1985] SMITH, B.C. (1985) Prologue to "reflection and semantics in a procedural language" In Brachman, R.J. and Levesque, H.J. (Eds.), *Readings in Knowledge Representation*, 31-40. Morgan Kaufman, California.

[Smithers, 1992] SMITHERS, T. (1992). Taking eliminative materialism seriously - a methodology for autonomous systems research. AI Memo 92-10, AI-Lab, Vrije Universiteit Brussel, Brussels, Belgium.

[Smithers et al., 1990] SMITHERS, T., CONKIE, A., DOHENY, J., LOGAN, B., MILLINGTON, K., AND TANG, M. (1990). Design as intelligent behaviour: An AI in design research programme. *Artificial Intelligence in Engineering Journal*, 5(2), 78–109.

[Sommerville, 1989] SOMMERVILLE, I. (1989). *Software Engineering (Third Edition)*. Addison Wesley, Bonn.

[Sorumgard et al., 1992] SORUMGARD, L.S., SINDRE, G. AND STOKKE, F. (1992) Experiences in reusable components classification In Constantopoulos, P. and Vassiliou, Y. (Eds), *Methods and Tools for Software Reuse*, 41-54. ERCIM Workshop Report ERCIM-92-W006, 1992

[Spivey, 1988] SPIVEY, J. (1988). *Understanding Z.* Cambridge, Cambridge University Press.

[Sprenger, 1991] SPRENGER, M. (1991). Explanaition strategies for KADS-based expert systems. DIAMOD Project Bericht Nr. 10, GMD, St. Augustin, Germany.

[Steels, 1990] STEELS, L. (1990). Components of expertise. *AI Magazine*, 11(2 (Summer)), 29–49.

[Taylor, 1992] TAYLOR, R. (1992). Risk management in knowledge-based systems development. KADS-II Report KADS-II/T2.1.2/WP/TRMC/003/1.0, Touche Ross.

[TERMIUM, 1992] TERMIUM (1992). Based de Donnees du Secreatriat d'Etat du Canada Based de Donnees du Secreatriat d'Etat du Canada, Ontario: OCHS

[Terpstra *et al.*, 1992] TERPSTRA, P., VANHEIJST, G., NORDBO, I., RAMPARANY, F., REICHGELT, H., SHADBOLT, N., & B.WIELINGA (1992). Ka process support in kew. deliverable ACK-UVA-A2-DEL-oo2-B, UvA.

[Terpstra, 1993] TERPSTRA, P. (1993) CommonKADS specific design decisions and their notation ESPRIT Project P5248 KADS-II/M7/UvA/043/1.1, University of Amsterdam. Deliverable DM7.2b.

[Thunem and Sindre, 1992] THUNEM, S. AND SINDRE, G. (1992) Development with and for reuse - Guidelines from the REBOOT project In Constantopoulos, P. and Vassiliou, Y. (Eds), *Methods and Tools for Software Reuse*, 41-54. ERCIM Workshop Report ERCIM-92-W006, 1992

[Tong *et al.*, 1988] TONG, X., HE, Z., & YU, R. (1988). A survey of the expert system tool ZDEST–2. In *Proceedings ECAI–88, Munich*, pages 113–118, London. Pitman.

[Turban, 1992] TURBAN, E. (1992). *Expert systems and applied artificial intelligence*. Macmillan, New York.

[Van de Velde, 1993] VAN DE VELDE, W. (1993). Generic Library Facility: Specification KADS-II/WPM/TM.8/DD/052/3.0, Vrije Universiteit Brussel, Brussels, Belgium.

[van Harmelen, 1989] VAN HARMELEN, F. (1989). Classification of meta-level architectures. In Jackson, P., Reichgelt, H., & van Harmelen, F., editors, *Logic-Based Knowledge Representation*, chapter 2, pages 13–36. MIT Press. Also in: *Meta-Programming in Logic Programming (META88)*, Abramson, H. and Rogers, M. H. (eds.), MIT Press, 1989, pp. 103-122.

[van Harmelen, 1992] VAN HARMELEN, F. (1992). Definable naming relations in meta-level systems. In Pettorossi, A., editor, *Proceedings of the Third Workshop on Meta-programming in Logic (META'92)*, Uppsala. Springer-Verlag.

[van Harmelen & Balder, 1992] VAN HARMELEN, F. & BALDER, J. R. (1992). $(ML)^2$: a formal language for KADS models of expertise. *Knowledge Acquisition*, 4(1). Special issue: 'The KADS approach to knowledge engineering'.

[van Harmelen *et al.*, 1992] VAN HARMELEN, F., AKKERMANS, J. M., BARTSCH-SPÖRL, B., BREDEWEG, B., COULON, C. H., DROUVEN, U., KARBACH, W., REINDERS, M., SCHREIBER, A. T., VINKHUYZEN, E., VOSS, A., & WIELINGA, B. J. (1992). Knowledge-level reflection: Specifications and architectures. ESPRIT Basic Research Action P3178 REFLECT, Deliverable R.2 RFL/UvA/III.2, REFLECT Consortium. Available from: University of Amsterdam, Social Science Informatics, Roetersstraat 15, 1018 WB, The Netherlands.

[van Harmelen *et al.*, 1991] VAN HARMELEN, F., BALDER, J. R., ABEN, M., & AKKERMANS, J. M. (1991). $(ML)^2$: A formal language for KADS conceptual models. Technical Report ESPRIT Project P5248 KADS-II/T1.2/TR/ECN/006/1.0, Netherlands Energy

Research Foundation ECN & University of Amsterdam. Deliverable D1.2.1. A shorter version is published in *Knowledge Acquisition Journal*, 1992.

[van Heijst et al., 1992] VAN HEIJST, G., TERPSTRA, P. WIELINGA, B. J., AND SHADBOLT, N. (1992) Using generalised directive models in knowledge acquisition In Wetter, T., Althoff, K., Boose, J., Gaines, B., Linster, M., and Schmalhofer, F. (Eds.), *Current Developments in Knowledge Acquisition: EKAW-92*, Berlin/Heidelberg, Springer Verlag.

[van Langevelde *et al.*, 1992] VAN LANGEVELDE, I. A., PHILIPSEN, A. W., & TREUR, J. (1992). Formal specification of compositional architectures. In Neumann, B., editor, *Proceedings ECAI'92, Vienna*, pages 272–276, Chichester. Wiley. To appear. Longer version available as: Report IR-282, Mathematics and Computer Science, Free University of Amsterdam.

[van Langevelde *et al.*, 1993] VAN LANGEVELDE, I., PHILIPSEN, A., & TREUR, J. (1993). A compositional architecture for simple design formally specified in DESIRE. In Treur, J. & Wetter, T., editors, *Formal Specification of Complex Reasoning Systems*. Ellis Horwood.

[Vanwelkenhuysen, 1991] VANWELKENHUYSEN, J. (1991). Troubleshooting digital printed circuit boards by means of an expert system. In *Proceedings of the 2nd European Test Conference*, Munich, Germany. vde-verlag.

[Vanwelkenhuysen, 1993] VANWELKENHUYSEN, J. (1993). *Participative design of industrial knowledge based systems*. PhD thesis, Vrije Universiteit Brussel, Brussels, Belgium. Also as VUB AI-Lab TR-93-1.

[vanWelkenhuysen & Rademakers, 1990] VANWELKENHUYSEN, J. & RADEMAKERS, P. (1990). Mapping knowledge-level analysis onto a computational framework. In Aiello, L., editor, *Proceedings ECAI–90*, pages 681–686, London. Pitman.

[Velthuijsen, 1992] VELTHUIJSEN, H. (1992). *The Nature and Applicability of the Blackboard Architecture*. PhD thesis, Rijksuniversiteit Limburg te Maastricht.

[Waern and Gala, 1993] WAERN, A. AND GALA, S. (1993). The commonkads agent model. ESPRIT Project P5248 KADS II KADS-II/M4/TR/SICS/004/V.1.0, Swedish Institute of Computer Science.

[Waern et al., 1993] WAERN, A., HOOK, K., AND GUSTAVSSON, R. (1993). The communication model. ESPRIT Project P5248 KADS II KADS-II/M3/SICS/TR/002/V1.0, Swedisch Institute of Computer Science, Stockhom, Sweden.

[Wetter, 1990] WETTER, T. (1990). First-order logic foundation of the KADS conceptual model. In Wielinga, B., Boose, J., Gaines, B., Schreiber, G., & van Someren, M., editors, *Current trends in knowledge acquisition*, pages 356–375, Amsterdam. IOS Press.

[Wielinga et al., 1992] WIELINGA, B., VAN DE VELDE, W., SCHREIBER, G., AND AKKERMANS, H. (1992). Towards a unification of knowledge modelling approaches. Technical Report ESPRIT Project P5248 KADS-II/T1.1/TR/UvA/004/4.0, University of Amsterdam, Free University of Brussels & Netherlands Energy Research Foundation ECN.

[Wielinga et al., 1992] WIELINGA, B., VAN DE VELDE, W., SCHREIBER, G., AND AKKERMANS, H. (1992). Towards a unification of knowledge modelling approaches. Technical Report ESPRIT Project P5248 KADS-II/T1.1/TR/UvA/004/4.0, University of Amsterdam, Free University of Brussels & Netherlands Energy Research Foundation ECN.

[Wielinga et al., 1993] WIELINGA, B., (ED) (1993). Expertise Model: Model Definition Document Technical Report ESPRIT Project P5248 KADS-II/M2/UvA/026/2.0, University of Amsterdam

[Williams et al., 1992] WILLIAMS, D. O., TOMLINSON, C., BRIGHT, C. K., AND RAJAN, T. (1992). The commonKADS quality viewpoint. Technical Report ESPRIT Project P5248 KADS-II/T2.2/TR/LR/0040/1.0, The KADS-II Consortium.

[Yost, 1992] YOST, G. R. (1992). Configuring elevator systems. Technical report, Digital Equipment Corporation, 111 Locke Drive, Marlboro, USA.

[Yourdon, 1989a] YOURDON, E. (1989a). *Managing the Structured Techniques*. Yourdon Press, Englewood Cliffs, New Jersey.

[Yourdon, 1989b] YOURDON, E. (1989b). *Modern Structured Analysis*. Prentice Hall, Englewood Cliffs, New Jersey.

Part IV

Appendices

Appendix A

Quality assurance and design decisions: an example

Vassilios Golfinopoulos, Walter Van de Velde

A.1 Introduction

This appendix identifies quality model trees for the blackboard architecture design model of the *Common*KADS approach. This exercise serves two purposes. First it illustrates the idea of coupling model decomposition and quality decomposition in such a way that quality assurance becomes feasible. This exercise is but one of those that need to be done for complete guidance but it mainly demonstrates the feasibility of the idea.

Secondly it contains the concrete results for the case of design using blackboard architectures. This entails a refinement of the CommonKADS design model for this specific case, a coupling with the standard quality models and concrete design guidelines derived therefrom.

The objective is to identify quality model trees that will reassure the quality of the blackboard architecture design model of the *Common*KADS approach. The contribution of this work consist of the quality model for a blackboard architecture.

A.2 The Blackboard Architecture

In this section we present a blackboard architecture which has three defining components: a global database called *blackboard* , independent knowledge sources that generate solution elements on the blackboard, and a control unit to control knowledge source activity [Nii, 1986]. The blackboard architecture is shown in Figure 2.1.

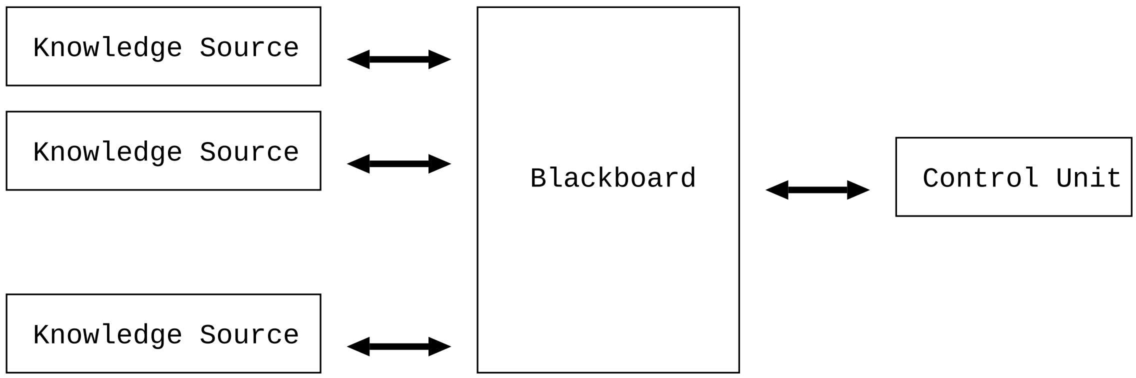


Figure 2.1 Blackboard architecture

A.2.1 Blackboard

A main component of a blackboard architecture is the blackboard, which serves two distinct functions [Velthuijsen, 1992, Nii, 1986, Turban, 1992]. First, it can be seen as a communications medium between the knowledge sources, and second it can be seen as a representation of the current state of the problem- solving process.

The blackboard in a blackboard architecture contains all the information that is relevant to the problem-solving process due to the interactions of the knowledge sources with the blackboard. In other words, the blackboard represents what has been achieved and also what next to be done to complete the problem-solving process.

All solution elements generated during the problem-solving process, as well as all the data elements are represented on the blackboard. These elements may trigger one or more knowledge sources since the blackboard elements indicate opportunities for future contributions to the problem-solving process. Thus, the blackboard can be seen as determining the applicability of knowledge sources. On the other hand information on the blackboard can be used to determine the desirability of applying a specific knowledge source.

In this way, the contribution of the knowledge sources result in proposing, extending, and modifying solution elements (hypotheses) that lead incrementally to a solution to the problem. An issue often seen as one of the characterizing features of the blackboard architecture is its ability to hold competing entries on the blackboard simultaneously and to pursue alternative processing threads at the same time [Nii, 1986].

An other characterizing feature that the blackboard serves is that of communication medium between the knowledge sources within the blackboard architecture. Knowledge sources in a blackboard architecture are independent since they are capable of independently executing a certain subtask, to solve a given subproblem, to generate a hypothesis, or to perform some other step in the problem-solving task of the complete blackboard system. They have to be coordinated in some way to solve a given problem cooperatively.

Another thing that has to be addressed, is the structure of the blackboard. The blackboard is not an unorganized collection of elements, but structured itself to facilitate interaction. The blackboard structure organizes solution elements along two axes, solution intervals and levels of abstraction. Different solution intervals represent different regions of the solution on some problem-specific dimension, while different levels of abstraction represent the different amounts of detail. Structuring a blackboard improves accessibility of a blackboard and efficiency in a blackboard system, and also improves understandability of information on the blackboard to users of a blackboard system [Velthuijsen, 1992].

The blackboard system Hearsay-II [Erman and Reddy, 1980] subdivided its blackboard into different levels which coincided with the different conceptual levels of the speech understanding problems. The levels contained hypotheses, regarding parameters of acoustic signals, segments of such signals, syllables, words, word sentences, phrases, and interpretation, respectively. This approach has been adopted in other systems as well, HASP/SLP [Nii et al., 1988], and Crysalis [Engelmore and Terry, 1979]. In Crysalis, for the first time, sets of blackboard levels were grouped to form panels.

Another approach presented in Hearsay-III [Balzer and Williams, 1980], POLIGON [Rice, 1986], and GBB [Corkill and Murray, 1986], extended the subdivision of the blackboard into levels by allowing the data to be structured via class hierarchies.

A.2.2 Knowledge sources

The domain knowledge needed to solve a problem is partitioned into knowledge sources that are kept separate and independent. The main goal of each knowledge source is to contribute information that may lead to a solution to the problem. A knowledge source takes a set of current information items on the blackboard and updates it regarding to its specialized knowledge.

The knowledge sources modify only the blackboard and only the knowledge sources modify the blackboard. Each knowledge source is responsible for knowing the conditions under which it can contribute to a solution.

The knowledge sources in a blackboard system can be seen as condition-action pairs. The condition part of a knowledge source specifies interesting situations for the knowledge source, and scans for opportunities to contribute to the overall problem-solving process. In many blackboard architectures, this scanning is facilitated by the use of events, which are representations of changes of information on the blackboard rather than of information itself.

The part of a condition part which specifies in which events a specific knowledge source is interested is usually called a trigger. Sometimes, additional blackboard conditions have to be checked in order to ensure that a knowledge source is indeed applicable. Thus, a typical condition part consists of two parts: one for specifying interesting blackboard state changes and one for specifying additional information.

Systems as Hearsay-II [Erman and Reddy, 1980] and BB1 [Hayes-Roth, 1985] use this subdivision, but systems as HASP/SIAP [Nii et al., 1988] and AGE [Nii and Aiello, 1979] rely only on events for deciding on the applicability of knowledge sources.

The action part of a knowledge source contains the code which actually can contribute to the problem-solving process, by modifying the blackboard. The action part can create new hypotheses or modify or delete existing hypotheses, in order to facilitate the contribution of other knowledge sources.

Whenever a specific knowledge source becomes applicable to appropriate information on the blackboard, the knowledge source and this information together form an instantiation for that knowledge source. It is possible that the same knowledge source may have different instantiations at the same time due to the situation where the same knowledge source is applicable to a number of events on the blackboard. Hearsay-II [Erman and Reddy, 1980] and BB1 [Hayes-Roth, 1985] represent instantiations via specific data structures. In BB1 these data structures are called knowledge-source activation record or, KSARs.

The philosophy of the blackboard architecture presupposes a subdivision of the problem-solving capabilities into knowledge sources. Thus, the blackboard architecture is a modular architecture [Velthuijsen, 1992]. Undeniably, one advantage of modularity is that knowledge sources can be added, removed, or exchanged easily.

A.2.3 Control unit

The blackboard control unit controls how the knowledge sources will respond opportunistically to changes on the blackboard [Nii, 1986]. The control unit may consist of a set of modules that monitor the changes on the blackboard and decide what actions to take next, which can be either the knowledge sources (which knowledge source to activate next), or the related hypotheses (which group of related hypotheses to pursue next) or a combination of both (which knowledge source to apply to which hypotheses).

Limited resources reinforce the need for a scheduler for mediating between competing knowledge sources. Sequential blackboard systems have a control unit executing a control loop, which in tum executes one or more knowledge sources, checks which knowledge sources have become applicable due to changes on the blackboard, and selects one or several of all applicable knowledge sources to be executed next. Figure 2.2, is adopted from [Velthuijsen, 1992], presents a control loop.

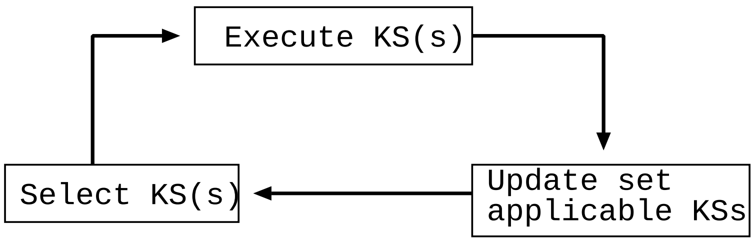


Figure 2.2 Example of control unit

Apart from these, in many blackboard systems, the blackboard panels and levels are organized in a hierarchy. When there is such a hierarchy the terms *top- down* processing and *bottom-up* processing can be defined:

Top-down processing starts at the highest level of the hierarchy and implements successive refinement of intermediate results on each lower level in order to arrive at the lowest level.

Bottom-up processing starts at the lowest level of the hierarchy and combines intermediate results on the lower levels to find intermediate results on higher levels. Bottom-up processing successive combining of intermediate results to reach a solution at the top level.

In [Nii, 1986], it is specified that *"since the knowledge sources apply opportunistically to changes on the blackboard, the solution is built one step at a time, and any type of reasoning step can be applied at each stage of solution formation. As a result, the sequence of knowledge source invocation is dynamic and opportunistic rather than fixed and preprogrammed".*

The reasoning step is said to be *data-driven*, when selection of activities is based only on currently available input and derived data. An *event-driven* processing emerges, when data changes instead of data itself is used for selection of activities.

The reasoning is said to be *goal-directed*, when the activities that are selected, are going to contribute to some explicitly stated goal or subgoal.

Data-driven control and goal-directed control are not defined in terms of hierarchies like top-down and bottom-up processing. Although, data-driven control often coincides with bottom-up processing and, similarly, goal-directed might coincide with top-down processing.

A.2.4 Refining design for blackboard architecture

This section deals with the study of the blackboard architecture for the architecture design of the design model. The overall goal is to identify the components of the blackboard architecture design and afterwards, in the next section, a quality model will be elaborated for the blackboard architecture design.

In other words, the refinement of the design model for blackboard architectures is presented. Actually, the goal is to consider the architecture design at a very abstract level, and applying the blackboard architecture in order to arrive at a completed architecture design model in terms of the em CommonKADS philosophy.

First of all, we have to take into account the architecture design model of the *Common*KADS approach, afterwards the high-level decomposition of the blackboard architecture, i.e., the blackboard, knowledge sources, control unit, and messages. Figure 2.4 demonstrates the high- level decomposition of the blackboard architecture design in terms of the em CommonKADS approach.

The architecture design is decomposed into a computational and an interaction architecture. We only focus on the computational architecture, since the interaction architecture is beyond the scope of this thesis. Thus, the computational architecture is further decomposed into computational objects, and computational methods. The computational objects that are used by the blackboard architecture are the blackboard, knowledge sources, and messages[1]

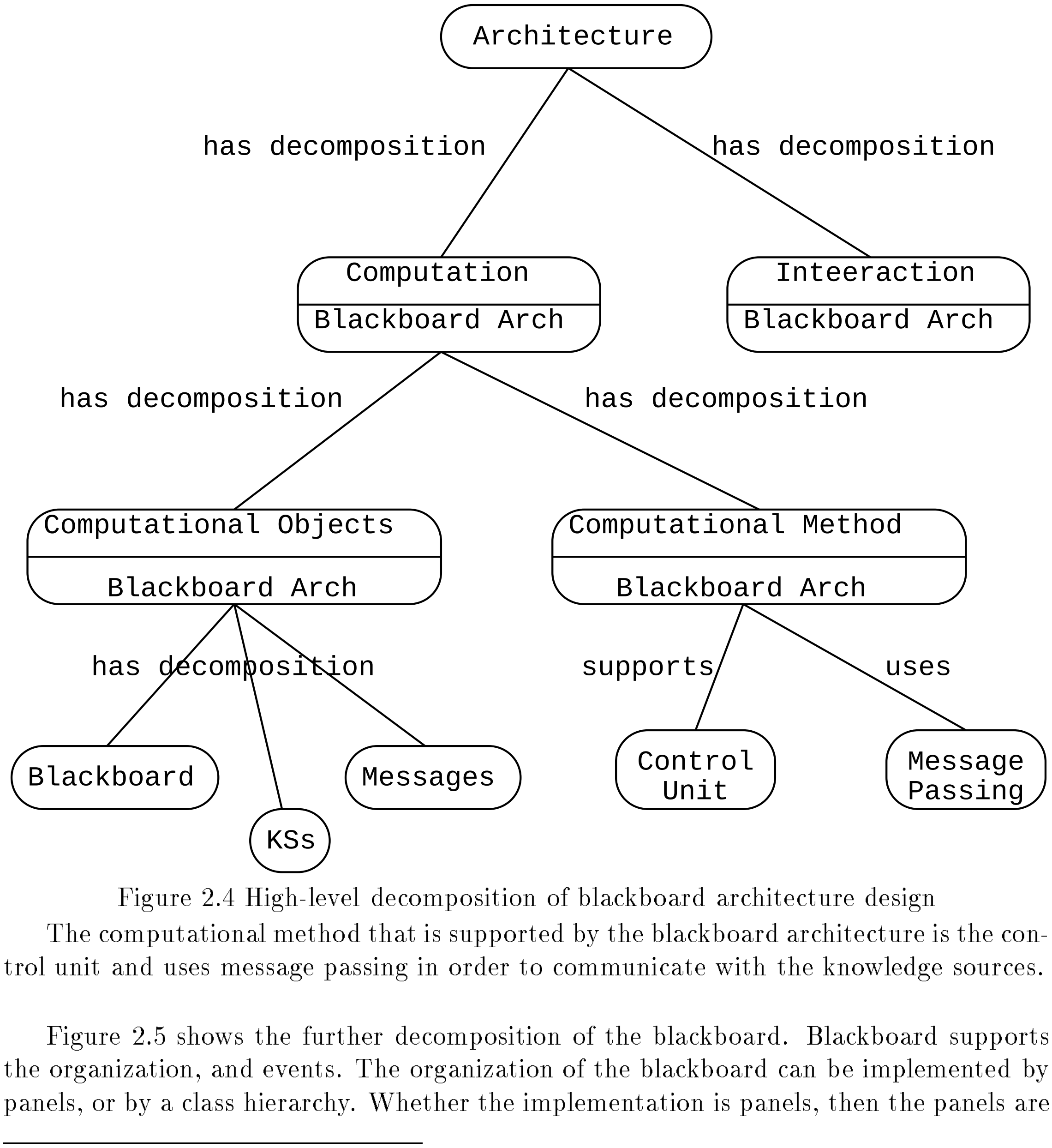


Figure 2.4 High-level decomposition of blackboard architecture design

The computational method that is supported by the blackboard architecture is the control unit and uses message passing in order to communicate with the knowledge sources.

Figure 2.5 shows the further decomposition of the blackboard. Blackboard supports the organization, and events. The organization of the blackboard can be implemented by panels, or by a class hierarchy. Whether the implementation is panels, then the panels are

[1]Actually the terms "messages" and blackboard "interactions" are equivalent. The term "messages" is used in order not to be confused with the interaction architecture of the design model.

decomposed into levels, which are themselves decomposed into hypotheses that belong to different levels, and moreover to different panels.

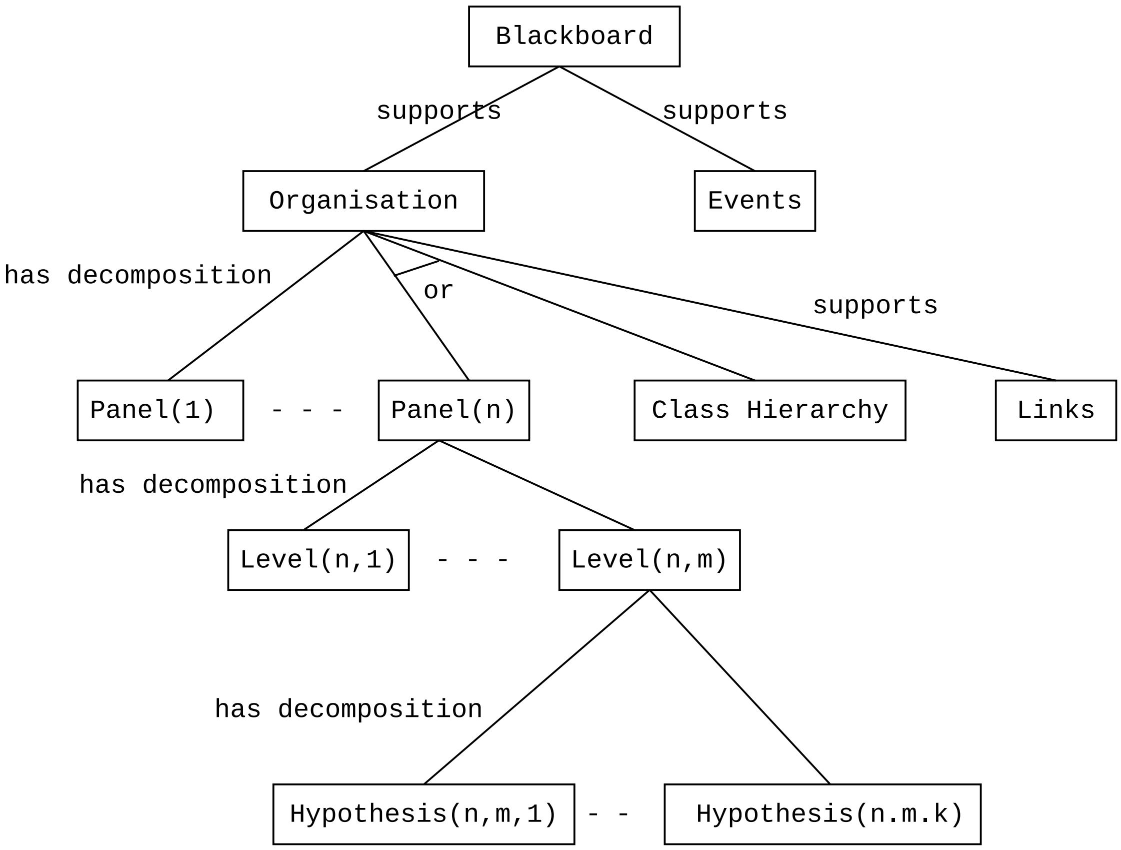


Figure 2.5 Blackboard decomposition

Besides, the organization of blackboard could support links whatever form can take, directed or labeled.

Figure 2.6 illustrates the low-level decomposition of the knowledge sources. Knowledge sources are decomposed into the KS descriptor, KS condition part and KS action part. Further decomposition is feasible since KS descriptor is decomposed into the KS identifier (or KS name), KS binding, and KS features. Additionally, the KS condition part is separated into KS trigger, and KS precondition. Furthermore, the KS action part is realized by messages.

Additionally, knowledge sources use data structures when become applicable, which are called instantiations or knowledge-source activation records (KSARs) [Hayes-Roth, 1985]. Only the term instantiation is going to be used.

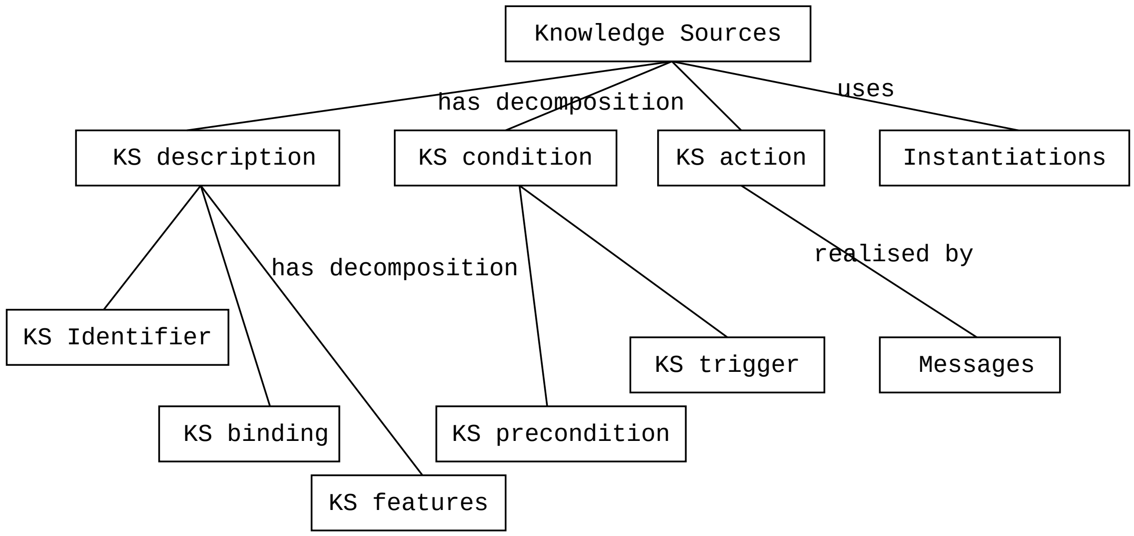


Figure 2.6 Knowledge sources decomposition

The messages are split into "read" and "write" messages. The read message is used by the knowledge sources in order to read information from the blackboard. The write messages are further decomposed into create, modify, and delete.

The write messages are used by the knowledge sources if they are to create a new hypothesis, or modify an existing hypothesis, or delete obsolete information on the blackboard. Figure 2.7 presents the messages decomposition.

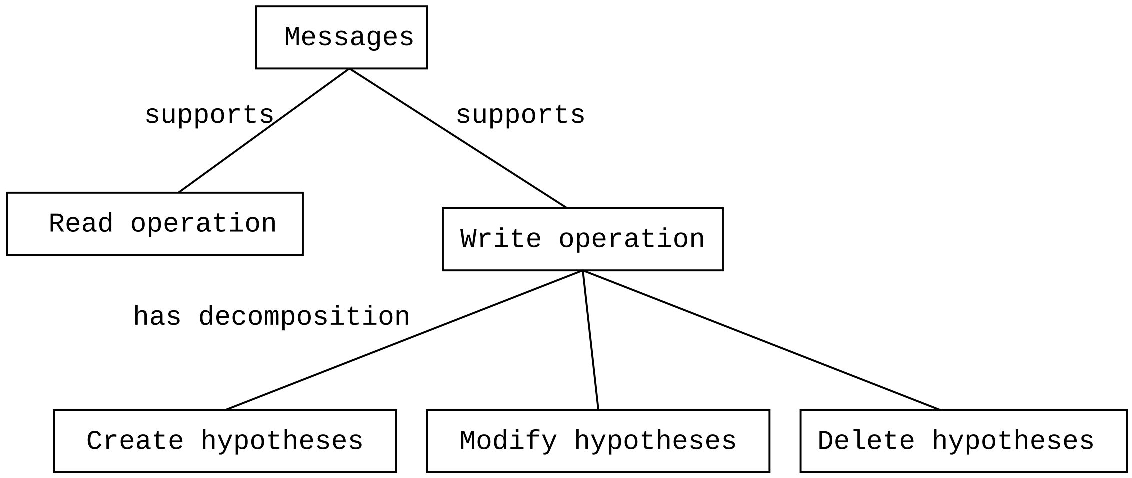


Figure 2.7 Messages decomposition

The control unit is the computational method that is supported by the blackboard architecture. It is decomposed into the basic cycle mechanism, matching mechanism, selection mechanism, and stop criterion. There are two alternative ways for implementing the basic cycle mechanism: KS cycle, and control cycle. Besides, the matching mechanism can be implemented through the KS-based or event-based technique. Moreover, blackboard-based and/without KS-based can implement the selection mechanism. Figure 2.8 presents the decomposition of the control unit.

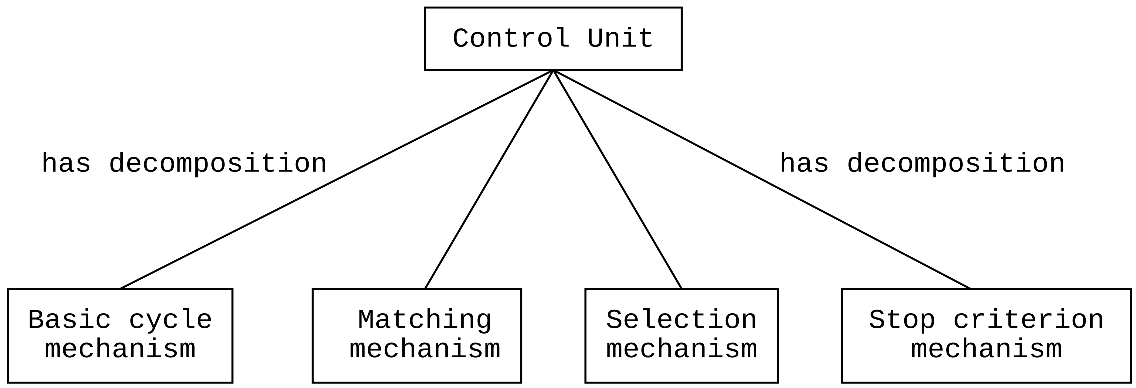


Figure 2.8 The decomposition of the control unit

A.2.5 Summary

This chapter presents the blackboard architecture. The blackboard, knowledge sources, and control unit consist the blackboard architecture. The blackboard not only serves as a communication medium between the knowledge sources, but also provides a representation of the current state of the problem-solving process. The knowledge sources respond opportunistically to changes on the blackboard. The control unit deals with situations like, which knowledge source can be executed, which blackboard data need to be processed next, selection of appropriate actions, and supervising the execution of knowledge sources. The top-down and bottom- up processing are defined, and the opportunistic problem-solving is examined.

A.3 Deriving design guidelines for a blackboard architecture

So far, we have focused on the refinement of the architecture design for blackboard architectures of the design model, starting with the top-level decomposition and during refinement we arrive at a low-level decomposition of the blackboard architecture, which designates the low-level components that constitute the blackboard architecture.

Generally speaking, within the boundaries of available resources, an objective from the project management point of view, is the quality of the project. Thus, the states of the (architecture) design must enable one to bind precise quality characteristics that will stipulate the preservation of the quality objective of the overall project. Generalizing this idea, a quality model has to be evaluated and then afterwards, must be measurable.

This section tackles the identification of the closest possible correspondence between architecture design model and product quality, in other words, the mapping of quality characteristics with the components of blackboard architecture design, resulting quality model trees. Thus, the goal is to consider on the one hand the refinement of the blackboard architecture design and on the other the quality characteristics, adopted from [ISO9126, 1990]. The six top-level quality characteristics, defined in [ISO9126, 1990], are:

- Functionality
- Usability
- Maintainability
- Efficiency
- Reliability
- Portability
- Reliability

A.3.0.1 Functionality

In [ISO9126, 1990]. functionality is defined as follows:

"A set of attributes that bear on the existence of a set of functions and their specified properties. The function are those that satisfy states or implied needs ".

Functionality is a top-level characteristic of the quality model. In [ISO9126, 1990], it is specified that in order to measure a top-level quality characteristic, a decomposition into low-level characteristics, is needed. Therefore, functionality is decomposed into suitability, accuracy, interoperability, compliance, and security. Their definitions are adopted from [ISO9126, 1990]:

Suitability "Attribute of software that bear on the presence and appropriateness of a set of functions for specified tasks. "

Accuracy "Attribute of software that bear on its ability to interact with specified systems.

Compliance "Attributes of software that make the software adhere to application related standards or conventions or regulations in laws and similar prescriptions. "

Security "Attributes of software that bear on it ability to prevent unauthorized access whether accidental or deliberate, to programs and data. "

The architecture specification has an impact on the functionality of the product. The blackboard architecture must be such that has an impact on the accuracy of functionality, therefore, the blackboard, KSs, control unit have an impact on the accuracy.

In a more precise manner, the events, i.e., the blackboard changes that are represented on the blackboard must be correct. The decision 1 specifies the possibility of implementing links, that has an impact on the augmentability of the product, since it gives the opportunity of using different hierarchies on the blackboard. The definition of correctness, augmentability, is adopted from [TERMIUM, 1992], is as follows:

Correctness "The ability of the software to produce specific outputs when given the specific inputs, and the extent to which a program satisfies its specifications and fulfills the user's mission. "

Augmentability "The extent at the code can easily accommodate expansion of computational functions within components or data storage requirements. "

The components of the KS descriptor and KS condition, which are the KS binding, KS features, KS precondition, and KS trigger must be specified concerning the accuracy of the architecture. T he same stands for the KS action as well. The definition of specificity, is adopted from [TERMIUM, 1992], is as follows:

Specificity "A program expressed in a computer code designed to solve a specific class of problems. "

The basic cycle must be correct that no impasse can occur in the inference cycle. For the matching mechanism, the decision 2 indicates that the amount of matching activities depends on the matching paradigm that it is going to be used. The decision 3 indicates whether the selection mechanism is going to be present or not. Actually, this decision depends upon two factors. Systems that implement a separate selection mechanism have better performance than systems that it is included in their basic cycle. The second factor, designates whether the system will be sequential or parallel.

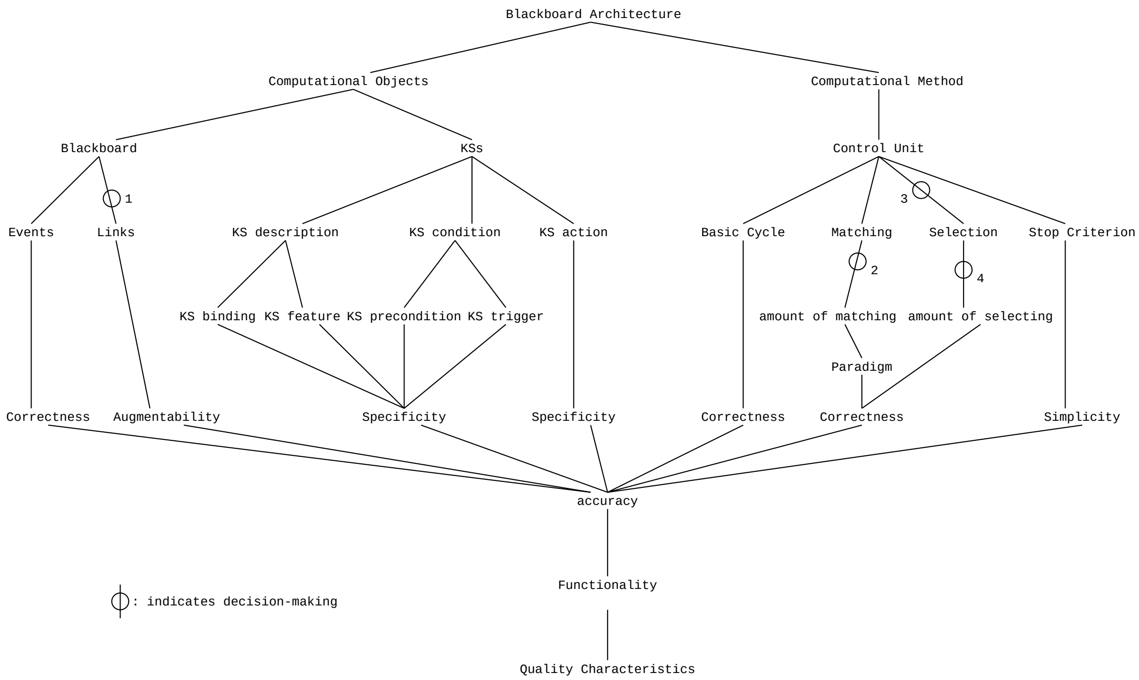


Figure 3.1 Functionality Model Tree

The decision 4 indicates that the amount of selecting activities must be less or equal to the amount of matching activities. Simplicity must underline the stop criterion in order to be easily detected. The definition of simplicity, is adopted from [TERMlUM, 1992], is as follows:

Simplicity "Implementation of functions in the most understandable manner, usually requiring avoidance of practices that increase complexity. "

The quality model tree for functionality and accuracy is illustrated in figure 3.1. Direct connections between other sub-characteristics of functionality such as suitability, interoperability compliance, and security, with blackboard architecture design can not be found. Generally speaking, these quality sub-characteristics should rule the overall blackboard architecture design.

A.3.0.2 Usability

Usability is another top-level quality characteristic. In [ISO9126, 1990], usability is defined as follows:

"A set of attributes that bear on the effort needed for use on the individual assessment of such use by a stated or implied set of users. "

In [ISO9126, 1990] usability is decomposed into understandability, learnability, operability. Their definitions are adopted from [ISO9126, 1990]:

Understandability "Attributes of software that bear on the users' effort for recognizing the logical concept and its applicability. "

Learnability "Attributes of software that bear of the users' effort for learning its application (e.g. operational control, input, output). "

Operability "Attributes of software that bear on the users' effort for operation and operation control. "

Figure 3.2 illustrates the quality model tree for usability and understandability.

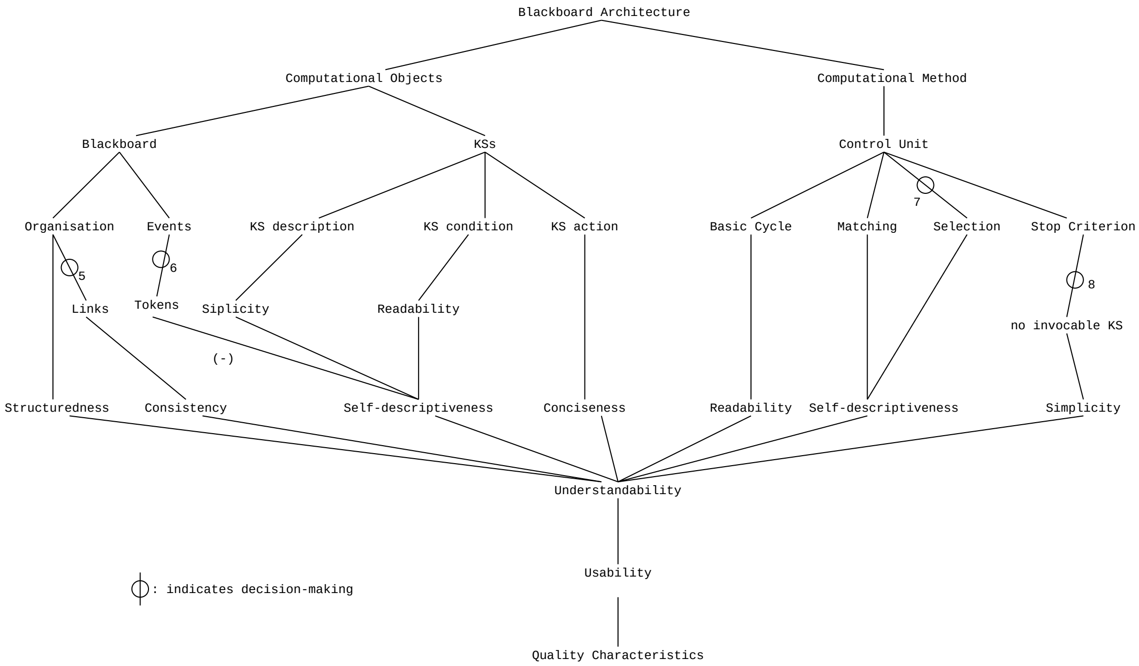


Figure 3.2 Usability Model Tree.

The blackboard architecture specification has an impact on the usability of the product. More precisely, the blackboard, KSs and control unit have an impact on understandability of usability.

Whatever implementation of the organization of the blackboard is found, the organization itself must be ruled by structuredness. The defined definition of structuredness, is adopted from [TERMIUM, 1992], is as follows:

Structuredness "The extent that the design possesses a definite pattern within its interdependent parts, and has proceeded in an orderly and systematic manner. "

The decision 5 indicates that the realization of links facilitates in consistency maintenance of the overall organization of the blackboard. The decision 5 indicates whether the implementation of events is going to be done through tokens. The main disadvantage of this realization is that tokens necessitate a clear a priori understanding of which events are relevant. The definitions of consistency and self-descriptiveness, adopted from [TERMIUM, 1992], are as follows:

Consistency "Attribute of software that provides uniform design and implementation techniques and notation. "

Self-descriptiveness "The extent that the code listing contains enough information for a reader to determine or verify its objectives, assumptions, constraints, inputs, outputs, components, and revision status. Commentary and traceability of previous changes by transforming previous versions of code into non executable but available code provide for self-descriptiveness. "

Concerning self-descriptiveness, the KS descriptor should be underlined by simplicity, and KS condition by readability. The definitions of readability, and conciseness, adopted from [TERMIUM, 1992], are below:

Readability "The extent that the code's function and design can be easily understood by reading e.g., complex expressions having mnemonic variable names and parentheses even if they are unnecessary). "

Conciseness "The absence of redundant or excessive coding and the assurance that the required functions are implemented with a minimum amount of coding. Conciseness implies that the program is not excessively fragmented and that the same sequence of coding is not repeated in numerous places. "

The KS action must be considered since only relevant blackboard modifications should be mentioned. Clearly, the control cycle has an impact on understandability of usability. Therefore, the basic cycle must be readable, and the matching, selection mechanisms should be self-descriptive. Decision 7 deals with the possibility of implementing the selection mechanism as a separate component. Decision 8 deals with the simplicity of the stop criterion which is reinforced by the possibility of the system to halt when there are no invocable KSs.

Direct connections between other sub-characteristics of understandability, such as learnability, and operability, with blackboard architecture design can not be found, but these sub-characteristics should rule the overall architecture design.

A.3.0.3 Maintainability

Maintainability is defined in [ISO9126, 1990]. The definition is adopted from [ISO9126, 1990], and is as follows:

"A set of attributes that bear on the effort needed to make specified modifications. "

Maintainability is decomposed into analyzability, changeability, stability, and testability. The definitions are adopted from [ISO9126, 1990].

Analyzability "Attributes of software that bear on the effort needed for diagnosis of deficiencies or causes of failures, or for identification of parts to be modified. "

Changeability "Attributes of software that bear on the effort needed for modification, fault removal or for environmental change. "

Stability "Attributes of so.ware that bear on the risk of unexpected effect of modifications. "

Testability "Attributes of software that bear on the effort needed for validating the modified software. "

Figure 3.3 presents the quality tree for maintainability. The architecture specification has an impact on the maintainability of the product. Especially, the blackboard, KSs, control unit have an impact on analyzability, changeability, and testability.

Decision 9 specifies the possibility of using an hierarchy for the blackboard organization, resulting in structuredness and correctness of the organization. The structuredness must be analyzable, changeable, testable in terms of the maintainability of the blackboard architecture. Decision 9 indicates the implementation of links, which results in maintaining the consistency, although the main disadvantage of the implementation is the complexity of the code. Traceability of links reassures the correctness of the function that is supposed to operate. In other words links should be analyzable, changeable, and testable.

Decision 11 has to do with the implementation of the hypotheses through data types. The advantage of this choice is the possibility of modifying, or expanding the functionality of the system. (For example, implementation of dynamic type checking.)

Whether the events are implemented by tokens, indicated by decision 12, results in reducing the self-descriptiveness of events, and further in reducing the maintainability of the product.

The KS descriptor should be general, concerning the analyzability and changeability. Besides, the modularity of the KS condition improves the flexibility of the KSs, and on the other hand the maintainability.

The KS action should be considered, and the consistency should underline the basic cycle, concerning the maintainability. The correctness of the matching, and selection mechanism must be analyzable, testable. The changeability is connected with the matching, selection mechanism in case of changing the paradigm that is used. Decision 13 indicates that in case of a KS-based selection mechanism the complexity of the code is increased. Finally, the simplicity of the stop criterion should be taken into consideration, since it must be analyzable, testable and may be changeable.

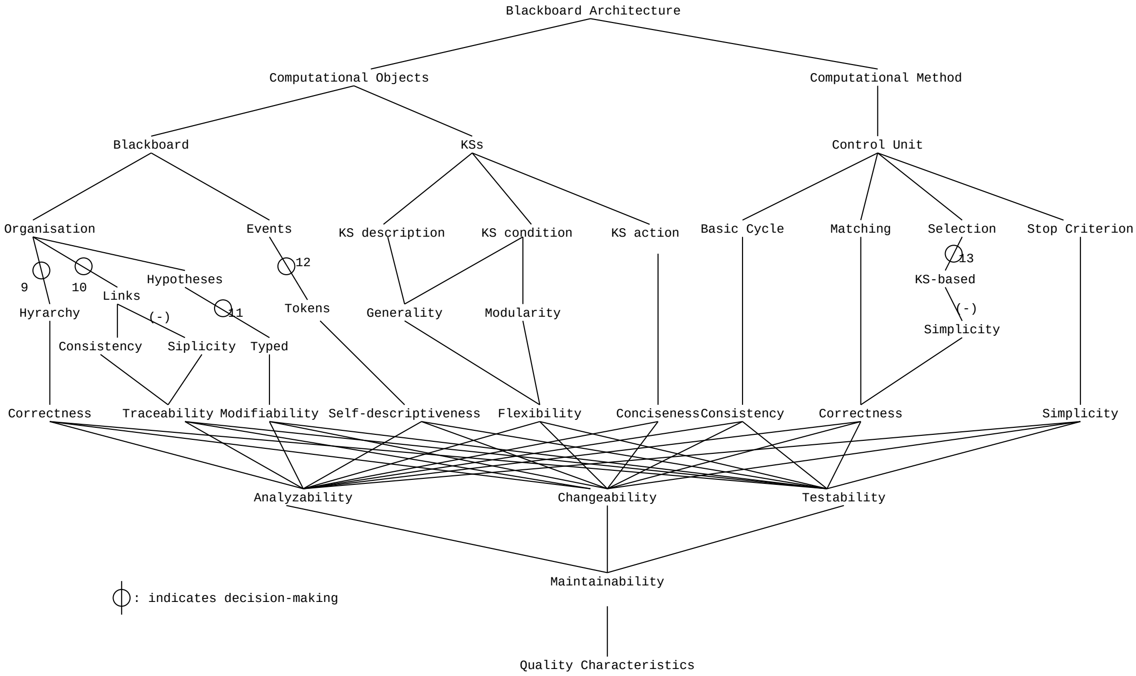


Figure 3.3 Maintainability Model Tree

Direct connection between stability and blackboard architecture design can not be found, but it should rule the overall architecture design. The definitions of modifiability, flexibility, generality, and modularity, are adopted from [TERMIUM, 1992], are as follows:

Modifiability "Characteristics of the design and implementation that facilitate incorporation of changes, once the nature of the desired change has been determined. "

Flexibility "Characteristics of the design and implementation that are concerned with the ability to adjust or adapt to change. "

Generality "A program expressed in a computer code designed to solve a class of problems, or specializing on a specific problem when appropriate parametric values are supplied."

Modularity "The extend to which software is composed of discrete components such that a change to one component has minimal impact on other components. "

A.3.0.4 Efficiency

Another top-level quality characteristic is efficiency, as defined in [ISO9126, 1990]. The definition of efficiency, is adopted from [ISO9126, 1990], is as follows:

"A set of attributes that bear on the relationship between the level of performance of the software and the amount of resources used, under stated conditions. "

Efficiency is decomposed into time behavior, and resource behavior, as specified in [ISO9126, 1990]. The definitions of the sub characteristics of efficiency, are adopted from [ISO9126, 1990], are as follows:

Time behavior "Attributes of software that bear on the response and processing times and on throughput rates in performing its function. "

Resource behavior "Attributes of software that bear on the amount of resources used and the duration of such use in performing its function. "

The architecture specification has an impact on the efficiency of the product. Moreover, the blackboard architecture has an impact on the resource behavior and on time behavior as well. The quality model for efficiency and resource behavior is presented in Figure 3.4, while the quality model for efficiency and time behavior is illustrated in Figure 3.5, and are treated independently.

The storage efficiency and the accessibility of the organization of the blackboard and the events increases the efficiency of the product in terms of resource efficiency. The decision 14 indicates that in case of using a class hierarchy for the organization of the blackboard the storage efficiency is increased since class hierarchy allows storage of blackboard entities in more specialized regions than possible with the use of blackboard levels and panels.

The applicability of the KSs is related to the correctness of the KS features, KS condition, as well as to the KS-binding. Also, the execution efficiency of the KS action increases the efficiency of the resource behavior.

The basic cycle must be robust such as that no impasse can occur in the inference cycle. The decision 15, 16 designates that in case of using an event-based, and a blackboard-based mechanism for matching and selecting respectively, the applicability of the KS trigger should be considered. Otherwise, the accessibility of the organization must be taken into account. Finally, the accessibility of stop criterion can influence the efficiency of the product.

The architecture specification has an impact on the time behavior of the efficiency. The decision 17 indicates that in case of using an hierarchy on the blackboard organization either in panels, levels or hypotheses, the accessibility of the organization is increased. The decision 18 specifies that the use of links can facilitate the access on the blackboard. The decision 19 denotes that when the implementation of events is done by tokens, events can be more efficiently matched with conditions.

The specificity of KS descriptor, and KS condition along with the accuracy of the KS action can increase the efficiency of the system. The decision 20 specifies that in case of including messages that can delete hypotheses that have become false, and/or modify existing hypotheses on the blackboard, increases the storage efficiency and the time efficiency on the other hand.

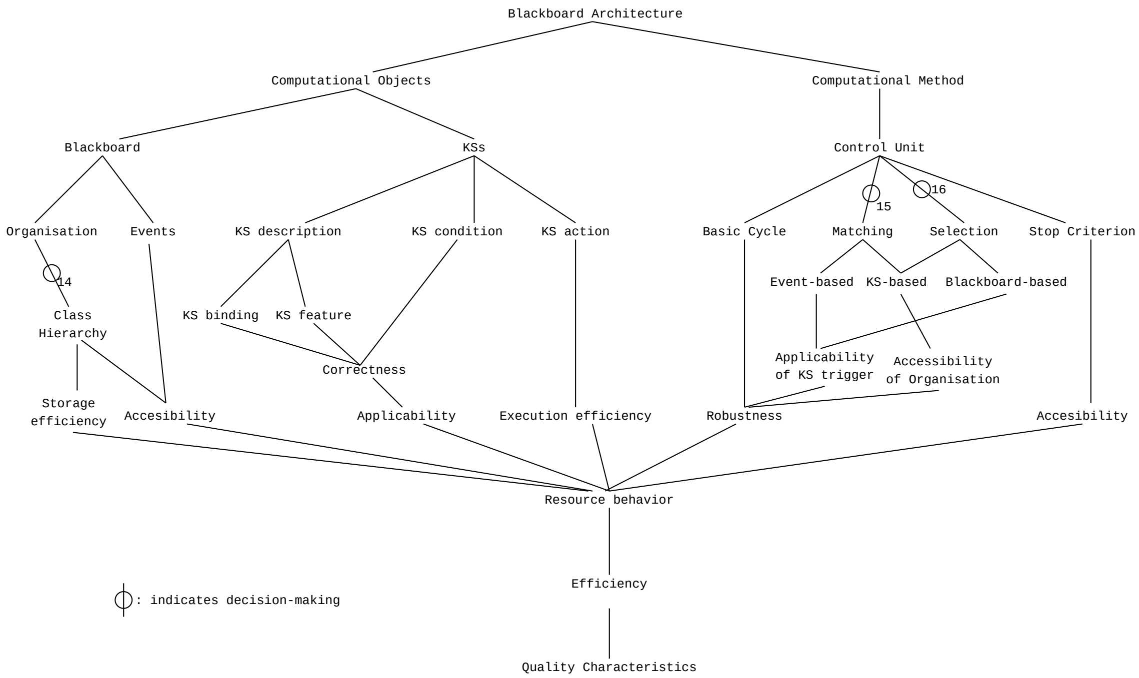


Figure 3.4 Resource Efficiency Model Tree

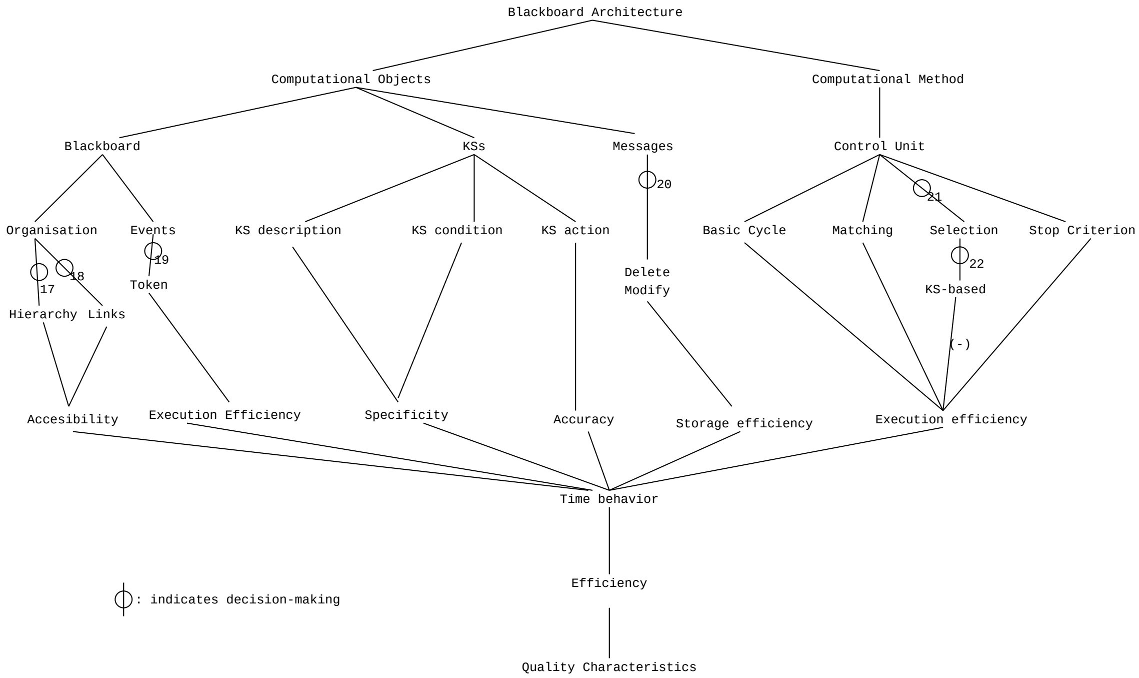


Figure 3.5 Time Efficiency Model Tree

The execution efficiency of the control cycle, matching mechanism, and stop criterion strongly are related to the time efficiency of the system. The decision 21, 22 specify that a separate selection mechanism gives better performance, but a KS-based selection mechanism reduces the execution efficiency since in addition to blackboard conditions that must be checked, KS conditions have to be considered as well.

The definitions of storage efficiency, accessibility, applicability, execution efficiency, robustness, accuracy, are adopted from [TERMIUM, 1992], are given below:

Storage efficiency "The extend to which software performs its intended functions with a minimum consumption of storage resources. "

Accessibility "The extend to which software facilitates selective use or maintenance of its components. "

Applicability "The extend to which a part of a component of the software declares the domain of objects over which the component can be applied. "

Execution efficiency "The extend to which software performs its intended functions with a minimum consumption of computing resources. "

Robustness "The extent that the software can continue to perform despite some violation of the assumptions in its specification. Robustness, implies, for example, that the program will handle inputs or intermediate calculated variables that are out of range or in different format or type than specified, without degrading the performance of functions not dependent on the inputs or variables. "

Accuracy "The extent that the software's outputs are sufficiently precise to satisfy their intended use."

A.3.0.5 Reliability

Reliability is defined in [ISO9126, 1990] as a top-level characteristic. The definition, is adopted from [ISO9126, 1990], is given below:

"A set of attributes that bear on the capability of software to maintain its level of performance under stated conditions for a stated period of time. "

In [ISO9126, 1990], reliability is decomposed into maturity, fault tolerance, and recoverability. The definitions, are adopted from [ISO9126, 1990], are given below:

Maturity "Attributes of software that bear on the frequency of failure by faults in the software. "

Fault tolerance "Attributes of software that bear on it ability to maintain a specified level of performance in cases of software faults or of infringement of its specified interface. "

Recoverability "Attributes of software that bear on the capability to re-establish its level of performance and recover the data directly affected in case of a failure and on the time and effort needed for it. "

Figure 3.6 illustrates the quality model tree of the reliability of the product. The blackboard architecture specification has an impact on the reliability, and more precisely on the fault tolerance and recoverability of the product.

The use of links as indicated in the decision 23, improves the consistency of the blackboard organization, which in turn increases the fault tolerance, recoverability of the reliability of the system. Besides, the simplicity of the events can increase the reliability.

The accuracy of the KS descriptor, KS action, along with the modularity of the KS conditions increase the reliability. The decision 24 specifies that in case of using messages for deleting, and modifying hypotheses, then the blackboard provides a correct representation of the current state of the problem-solving process of the blackboard system.

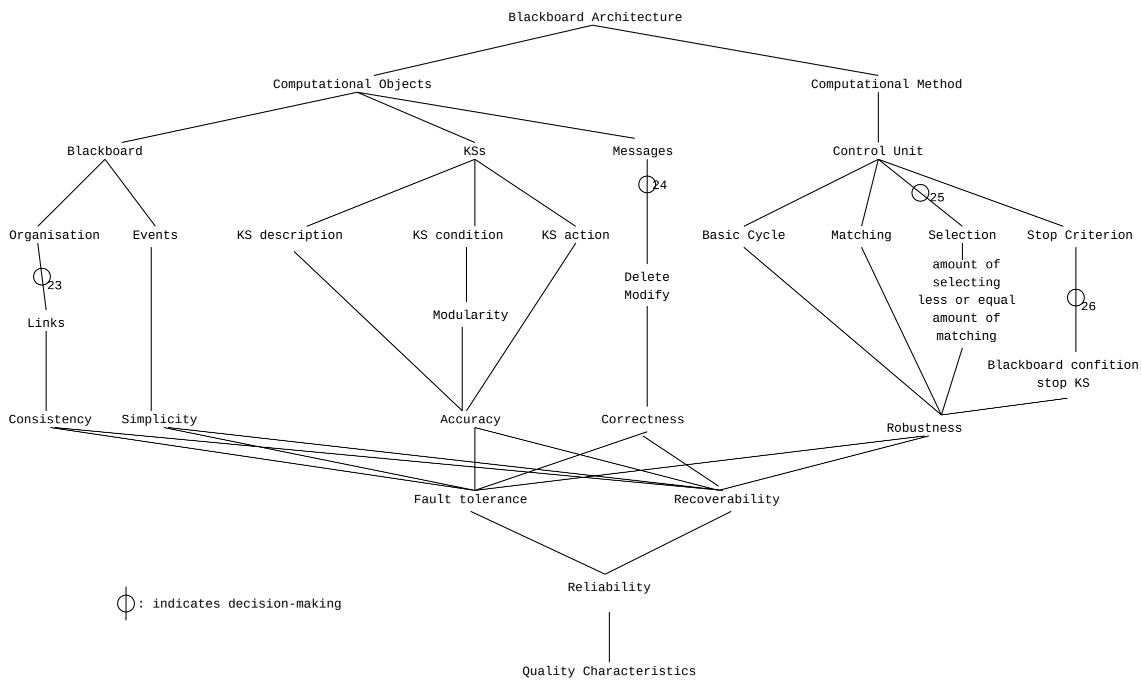


Figure 3.6 Reliability Model Tree

Furthermore, the reliability of the system is increased by the robustness of the control unit. The decision 25 indicates that an independent selection mechanism can improve the robustness of the control unit, together with a possible stop criterion in terms of a blackboard condition or a stop KS (decision 26).

Maturity should rule the overall architecture design since there is no direct connection with blackboard design.

A.3.0.6 Portability

Finally, portability is another top-level quality characteristic, as it is indicated in [ISO9126, 1990]. The definition, is adopted from [ISO9126, 1990], is as follows:

"A set of attributes that bear on the ability of software to be transferred from one environment to another. "

Portability is decomposed into adaptability, installability, conformance, replaceability. The definitions, are adopted from [ISO9126, 1990], are given below:

Adaptability "Attributes of software that bear on the opportunity for its adaptation to different specified environments without applying other actions or means than those provided for this purpose for the software considered. "

Installability "Attributes of software that bear on the effort needed to install the software in a specified environment. "

Conformance "Attributes of software that make the software adhere to standards or conventions relating to portability. "

Replaceability "Attributes of software that bear on opportunity and effort of using it in the place of specified other software in the environment of the software."

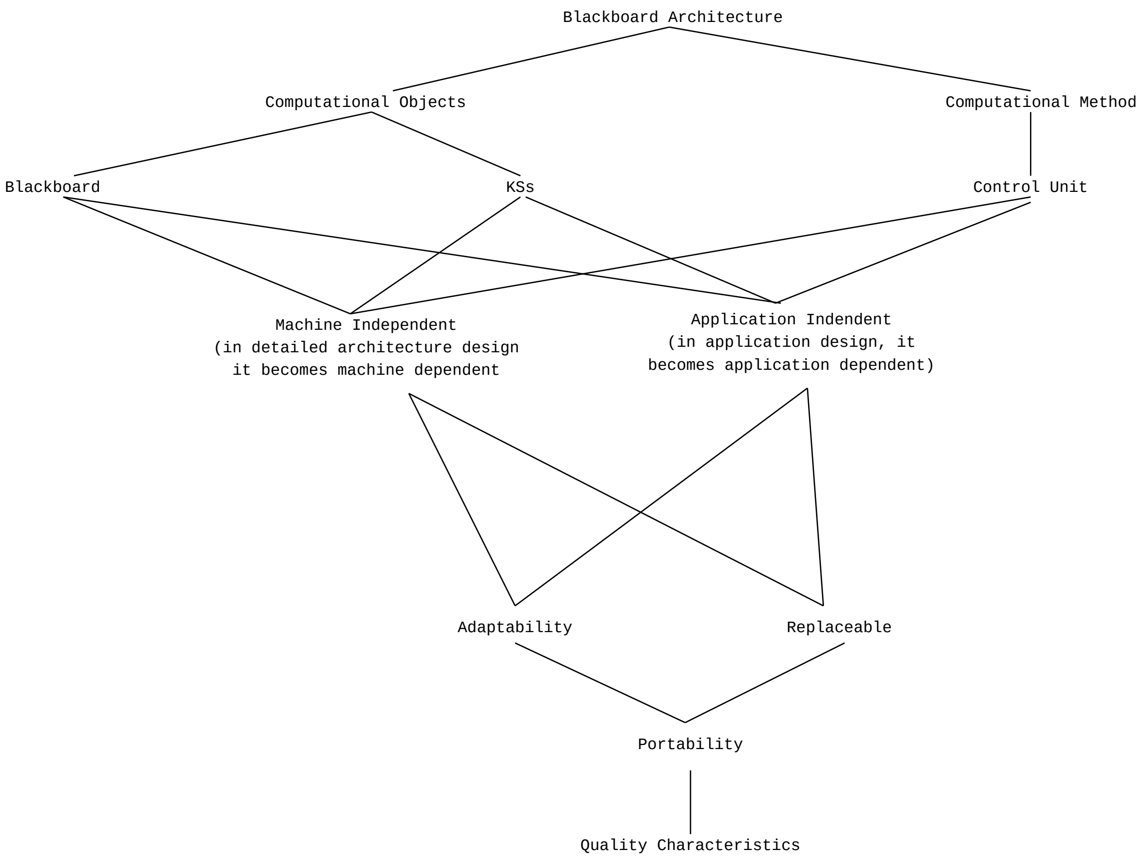


Figure 3.7 Portability Model Tree

The architecture specification has an impact on the portability of the product, and more precisely on the adaptability and replaceability of the system. The quality model tree is illustrated in Figure 3.7.

The main components of the blackboard architecture, the blackboard, KSs, and control unit, should be machine, and application independent in order to increase the adaptability and the replaceability of the system.

However, it must be pointed that the machine independent only stands for the architecture specification, since in detailed architecture design it becomes machine dependent. Exactly, the same situation can be considered for the application design since the architecture becomes application dependent.

The definitions of application independent, and machine independent, are adopted from [TERMUM, 1992], are given below:

Machine independent "The ability of the software to be unaffected by changes to the computer hardware or peripheral equipment. For independence, coding directly related to a specific hardware device should be minimized, isolated, and identified. "

Application independent "The extend to which an architecture can be used to develop applications by adding domain-dependent information. "

Other quality sub-characteristics of portability are the installability, conformance. Since direct connections between these sub-characteristics and blackboard design can not be found, they should only rule the overall architecture design.

Even it is beyond the scope of this thesis, beside the architecture specification, the detailed architecture design, generally speaking has an impact on the efficiency, maintainability, reliability, and portability of the product.

A.3.1 Identifying risks

In this section a list of risks is provided that may affect the need for the design model to be developed in a certain state. A risk is formulated whenever a decision has to be made may influence different quality characteristics in antithetical ways. In other words, a decision can improve a quality characteristic of the product, while at the same time may deteriorate another quality characteristic.

We are going to examine 9 situations where the decision may affect in antithetical ways different quality characteristics:

Design guideline 1

Situation: Speed and performance are important

Objective: Efficiency: $\Uparrow$
Accuracy of functionality: $\Uparrow$
Maintainability: $\Downarrow$

Risk: The specified requirements for the performance are not met

Target state: Specificity of KS condition

Rational: If the KS condition deals with a few cases, be specific, improves the time efficiency of the system concerning matching. If on the other hand, the KS condition deals with many cases, be general, improves the analyzability, changeability, which are sub-characteristics of maintenance.

Design guideline 2

Situation: Reliability is crucial

Objective: Reliability: $\Uparrow$
Maintainability: $\Downarrow$

Target state: Accuracy of the KS descriptor

Rational: If the KS descriptor is accurate (i.e., specific links to be addressed, variables to be involved), improves the reliability of the system. On the other hand, a general KS descriptor (i.e., many possibilities to address blackboard, all possible variables involved) can improve changeability, and possible testability sub-characteristics of maintainability.

Design guideline 3

Situation: Build a blackboard shell

Objective: Portability: $\Uparrow$
Suitability: $\Downarrow$

Risk: Application requirements are not met

Target state: Architecture specification must be application independent

Rational: If the architecture specification is application independent, improves the portability of the architecture design. On the other hand, the suitability, sub-characteristic of functionality, is deteriorated since there is no task orientation.

Design guideline 4

Situation: Build a blackboard shell

Objective: Portability: $\Uparrow$
Accuracy: $\Downarrow$

Risk: Architecture requirements are not met

Target state: Architecture specification must be machine independent

Rational: If the architecture specification is machine independent, improves the portability of the design. On the other hand, the accuracy of functionality of architecture design is deteriorated, since for example the degree of precision of calculated values can not be found.

Design guideline 5

Situation: Cruciality of termination

Objective: Reliability: $\Uparrow$
Understandability: $\Downarrow$
Maintainability: $\Downarrow$

Target state: Robustness of stop criterion

Rational: The implementation of stop criterion in a more robust way (i.e., blackboard condition, Stop KS) improves the reliability. On the other hand, a simplified implementation (i.e., no invocable KS) can also improve the understandability of usability, and maintainability.

Design guideline 6

Situation: Efficiency in matching

Objective: Efficiency: $\Uparrow$
Understandability: $\Downarrow$
Maintainability: $\Downarrow$

Target state: Implementation of events as tokens

Rational: Tokens can be more efficiently matched with conditions, but are must less flexible than more descriptive events, and necessitate a clear a priori understanding of which events are relevant.

Design guideline 7

Situation: Obviousness of operation

Objective: Understandability: $\Uparrow$
Maintainability: $\Downarrow$

Target state: Transparency of basic cycle

Rational: The transparency and readability improves the understandability of usability. On the other hand, the consistency of the basic cycle improves the maintainability. The above two situations are loosely related.

Design guideline 8

Situation: Knowledge sources efficiency is important.

Objective: Resource efficiency: $\Uparrow$
Time efficiency: $\Downarrow$

Target state: KS-based selection mechanism

Rational: A KS-based selection mechanism increases the resource efficiency, since it is more robust than a possible blackboard-based. On the other hand, it deteriorates the execution efficiency, and in addition the time efficiency of the product.

Design guideline 9

Situation: Speed and accuracy are important

Objective: Accuracy: $\Uparrow$
Time efficiency: $\Uparrow$
Maintainability: $\Downarrow$

Target state: Implementation of links

Rational: In case of implementing links, the accuracy of functionality, and the time efficiency are increased. On the other hand, the maintainability is decreased since the code complexity is increased.

A.3.2 Examples

Hearsay-II [Erman and Reddy, 1980], is a speech understanding system. Concerning the organization of the blackboard uses one panel, which consists of ordered levels, and every levels has time ordered hypotheses. The hierarchy of the levels and hypotheses, gives great accessibility to the blackboard, increasing the time efficiency of the system, the understandability. Furthermore, the use of directed links increases the consistency of the system, reliability, accuracy of functionality and time efficiency but on the other hand deteriorates the simplicity by increasing the code complexity, which may impede from the analyzability and changeability of the maintainability. The use of "modify" and "delete" messages increases the storage efficiency and the correctness of blackboard information, which has an impact on reliability and time efficiency. Events take the form of attribute-value pairs, which improves self-descriptiveness, and the understandability of usability.

The selection mechanism that is blackboard-based and KS-based and along with the correct number of selected activities increase the reliability of the system. On the other hand the combination of the selection techniques that is used increases the complexity of control unit. The choice of using a stop KS for the stop criterion gives robustness and reliability of the system.

GBB (Generic BlackBoard development system) [Corkill and Murray, 1986] is a blackboard framework. The use of a class hierarchy gives great accessibility to the blackboard, concerning the time efficiency, and also improves the storage efficiency. The use of labeled links, and ordered hypotheses (in case of GBB hypotheses are called units) provide efficient blackboard interaction mechanisms. The opportunities for implementing a control unit and KSs, result in a system which is generic. Eventually, since GBB is a blackboard framework, is application independent, which increases the portability of the framework.

A.3.3 Dependent-decisions chain

So far, we have examined decisions that may influence the quality characteristics of the product. There are cases where different qualities characteristics can be influenced in an opposite way. However, one may find decisions, concerning the implementation of the components of the blackboard architecture, that do not contribute to the quality model trees, in the sense that whatever choice we make the quality objective is preserved. The paradigm that will be used by the basic cycle (control cycle or KS cycle), matching mechanism (event-based or KS- based), the way of implementing the KS feature, etc., are examples of this kind of decisions that have to be taken into consideration. For this reason, the dependent-decisions chain is defined as follows:

A *dependent-decisions chain* can be defined as a chain consisting of dependent decisions which formulate an ordering.

One of the most important decisions in constructing the design model, is the number of processors that the system will accommodate. The decision concerning blackboard organization is application dependent. In case of sequential system, the decision of the selection mechanism and the implementation of blackboard organization will affect the decision for the KS feature, which in turn will affect the decision of the basic cycle, which in turn will affect the number of instantiations, which in turn will affect the KS binding, which will eventually affect the decision of stop criterion. Figure 3.8 gives a dependent-decisions chain that encloses the above idea. Moreover, in case of parallelism there is no constraints necessitate selection.

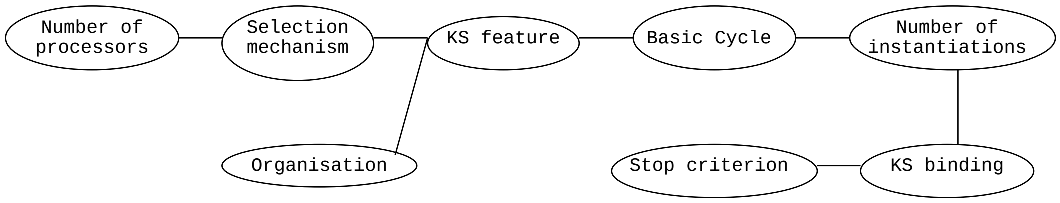


Figure 3.8

Suppose that the number of possible processor is n. Selection takes n from m applicable activities, where n $\leq$ m, based on KS features. The basic cycle will delay m-n activities until much later, therefore, currently applicable activities should be represented via instantiations or descriptors. The control cycle in order to have access to situation enabling an activity (m-n set), binding seems appropriate. When there is a control cycle, it is convenient to include a stop criterion.

It must be pointed out that the task of the application or the decision of the implementation of hypotheses, will affect the decision for the events, which in turn will affect the decision of the KS trigger. Figure 3.9 illustrates a possible dependent-decisions chain.

A special attention should be given in the decision of the control cycle, which is not application dependent, but dependents on the developing team. One possible criterion could be the experience on implementing control cycle or KS-based basic cycle.

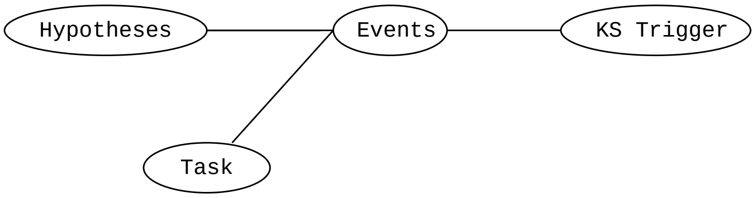


Figure 3.9

The decision of the basic cycle is very loosely related with the decision of matching mechanism. Theoretically speaking, a control-cycle mechanism should be followed by an event-based matching mechanism, and a KS-based basic cycle should be followed by a KS-based matching mechanism. However, practically speaking all possible combinations are possible and feasible.

For example, Hearsay-II [Erman and Reddy, 1980] is a sequential system, which uses a KS-based and blackboard-based selection mechanism. The selection mechanism needs to weigh the KS features of all the applicable KS in order to select only one, which will be executed. The control cycle has to delay the rest of the applicable activities, which are represented by frames. In order to activate a certain activity, the control cycle should have access to KS binding, which is represented as a frame as well. Finally, the stop criterion is implemented by a stop KS, which whenever is applicable will be selected, and the system will terminate.

Besides, the implementation of the hypotheses, events is done by attribute-value pairs. The KS trigger is implemented in the same way that events are implemented. Thus, there exist a kind of compatibility between the implementations of these components.

The same reasoning can be applied for BB1 [Hayes-Roth, 1985] which is another sequential blackboard system, and uses a KS-based and a blackboard-based selection mechanism. The organization of the blackboard consists of two panels, each of them consists of an ordered set of blackboard levels. The hypotheses are called units and are implemented as attribute-value pairs. The KS features also use attribute-value pairs. The basic cycle is a control cycle. The selection mechanism selects one activity, which will be executed. The KS binding indicates a special data structure representing an instantiation which is a KS activation record (KSAR), and when there are no more invocable KSARs, the system terminates. The events are represented as records, and the KS trigger is an event- based trigger.

Another example is CAGE [Aiello, 1986] which is a parallel system. It does implement a blackboard-based selection mechanism, even if it is not necessary if the number of selected activities are less or equal to the number of processors. The KS features consist of references to the blackboard panels and levels, links to be addressed, and a rule evaluator. The basic cycle is a control cycle. CAGE first select an event or a number of events and then proceed by executing all KSs that have a condition matching the selected event(s). The number of the total number of events that are matched may vary between only one and all. CAGE allows the user to specify the number of selected activities. CAGE provides a facility for binding of variables by the left-hand sides (LHSs) of the rules in the corresponding action part for use by other rules in the same action part. Also, the stop criterion is formulated in terms of the contents of the blackboard.

CAGE uses a class hierarchy, and a hierarchy of levels as well. The hypotheses, which are called nodes, are implemented by attribute-value pairs. Directed and labeled links are used, implemented as attributes. Events are implemented by tokens, and KS trigger uses a list of tokens.

A.3.4 Conclusion

Quality model trees specify the ways of how the components of a system have to be evaluated in order to reassure the product quality. Their role can been seen as control, since every decision-making and every possible non-optimal functioning, have an impact on the quality model trees, and furthermore on the overall product quality model.

This thesis is focused on the blackboard architecture design, and all the quality model trees are evaluated depending on this architecture. Generally speaking, quality models trees should be evaluated for the application, and platform design model as well, in order to have an overview of the product quality model.

After evaluating of quality model trees, the following step, is the quality metrics selection [Fenton, 1992]. The need exists to establish metrics that correlate to the characteristics of the software product. Thus, every quantifiable feature of software and every quantifiable interaction with its environment that correlates with a characteristics should be established as a metric.

Metrics can differ depending on the environment and the phases of the development process in which they are used. Metrics should be correlated to the user respective metrics, because the metrics from the user's view is more crucial.

Furthermore, rating levels have to be defined [Fenton, 1992] in order to allow software to be rated in accordance with the stated or implied needs. The result of metrics should be a measured value that is mapped on the scale. These scales have to be divided into ranges, if we are to show the different degrees of satisfaction.

Besides, the assessment criteria [Fenton, 1992] have to be defined. An assessment is an action of applying documented assessment criteria to a specific product model for the purpose of determining acceptance or release. To assess the quality of the product, the results of the evaluation of the different characteristics must be summarized.

Moreover, the actual measurement, the selected metrics are applied to the product model. The results are values on the scales of the metrics. Assessment is the final step, where a set of rated levels are summarized. The result is a statement of the quality of the product model. Then the summarized quality is compared with other factors such as time and cost. Finally, managerial decision will be made based on the managerial criteria. The result is a managerial decision on the acceptance or the rejection or on the release or no release of the product model.

A.3.5 Summary

This chapter deals the quality and the blackboard architecture. The blackboard architecture is applied to the *Common*KADS design model, resulting a low-level decomposition of the architecture. Quality models trees are defined, consisting of components of the blackboard architecture and quality characteristics, in order to specify the closest possible correspondence between the architecture design model and the quality model. The goal of the quality models trees is to reassure the quality of developing system, and give feedback since risks are identified, forming a list of guidelines. Dependent-decisions chains impose an ordering on the decision-making, leading to a more natural ways of developing a system.

Quality models trees must be evaluated for the application, and platform design. The next step is to find metrics, along with scales, and ranges corresponding to the different degrees of satisfaction in order to measure quality characteristics. An assessment is the last step, where the set of measured characteristics is compared with other aspects as time and cost. Finally, the result is a managerial decision on the acceptance or non-acceptance of the product model.

Appendix B

Example Implementation: Abstract & Specify

Guus Schreiber

This appendix contains the source code for the "abstract & specify" example. The implementation follows the structure-preserving principles defined in Ch. 10. Fig. B.1 shows the organisation of the modules found below.

B.1 Top-level module

```
:- use_module(['task-interpreter.pl']).

:- dynamic
    trace/1, trace/2.

:- assert(trace(task)).
:- assert(trace(inference)).

abstract_example(NewFindings) :-
    exec_task('Abstract new evidence',
        'new evidence' = [ temperature = 40, diastolic_pressure = 100 ],
        'new findings' = NewFindings).

specify_example(Observation) :-
    exec_task('Clarify finding',
         'finding'     = hypertension = present,
         'new observation' = Observation).
```

B.2 Task-level modules

B.2.1 Generic modules

Task interpreter

```
:- module('task-interpreter.pl', [exec_task/3]).
```

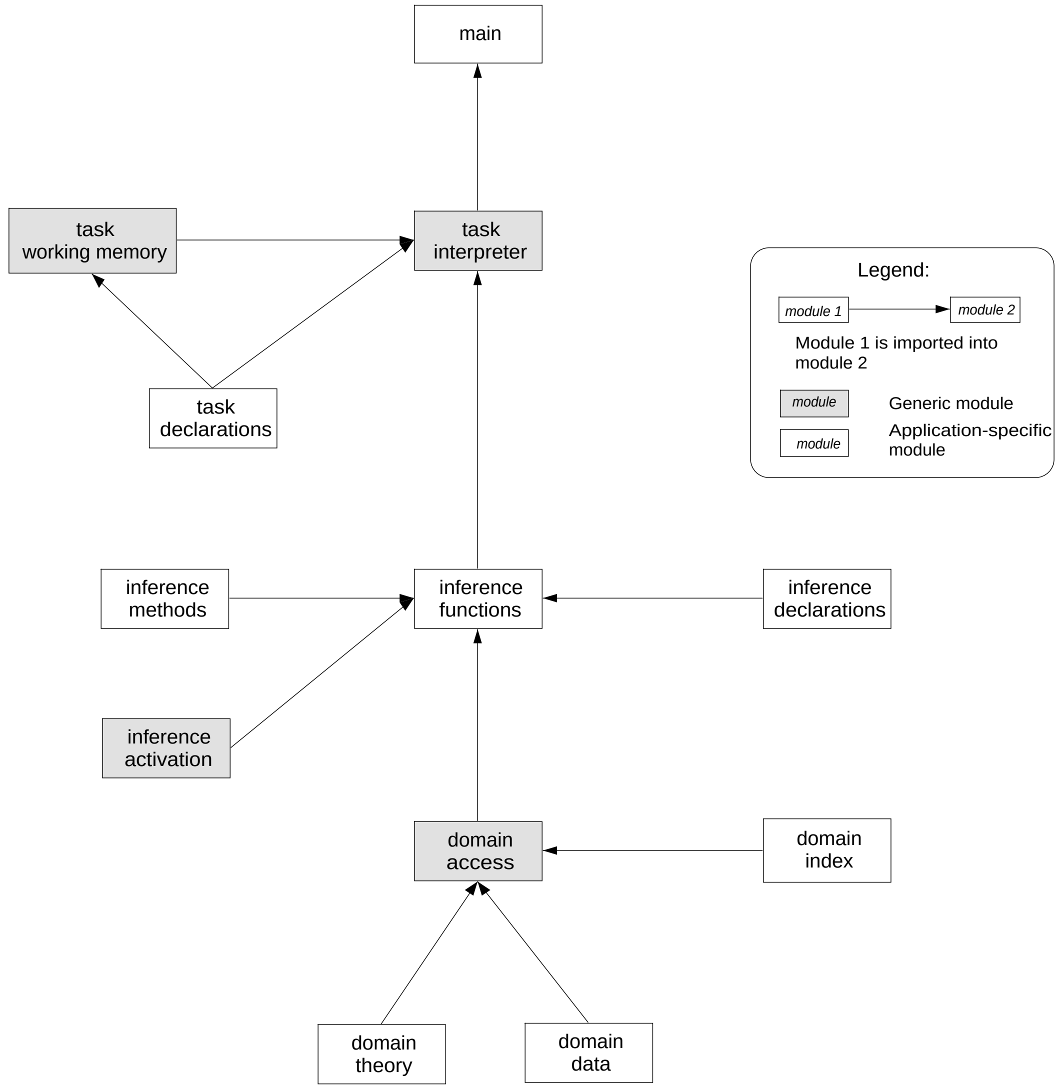


Figure B.1: Import relations between the Prolog modules.

```
:- use_module(
  [ 'task-declarations.pl'
  , 'task-working-memory.pl'
  , 'inference-functions.pl'
  ]).

%
% Task execution primitives
%

exec_task(Task, Input, Output) :-
    ignore(write_task(start, Task, Input)),
    task_interpreter(Task, Input, Output),
    ignore(write_task(end, Task, Output)).
```

```
task_interpreter(Task, Input, Output) :-
    init_task(Task, Input, Output),
    init_task_procedure(Task),
    once(task_procedure(Task)),
    retrieve_output(Task, Output).

init_task(Task, Input, Output) :-
    init(input, Task, Input),
    init(output, Task, Output),
    init(control_term, Task), !.

init(Type, Task, Arg) :-
    is_list(Arg), !,
    checklist(init(Type, Task), Arg).
init(input, Task, Name = Value) :-
    task_input(Task, Name),
    data_type(Name, DataType),
    data_operation(create, DataType, Name, Value).
init(input, Task, Name) :-
    task_input(Task, Name).
init(output, Task, Name = _) :-
    task_output(Task, Name),
    data_type(Name, DataType),
    data_operation(create, DataType, Name, []).
init(output, Task, Name) :-
    task_output(Task, Name).
init(control_term, Task) :-
    forall( control_term(Task, Name),
      ( data_type(Name, DataType)
      , data_operation(create, DataType, Name, [])
      )).

init_task_procedure(Task) :-
    (   clause(task_procedure(Task), _)
    ;   control_structure(Task, TaskStructure)
    ,   assert((task_procedure(Task) :- TaskStructure))
    ).

retrieve_output(Task, Output) :-
    is_list(Output), !,
    checklist(retrieve_output(Task), Output).
retrieve_output(Task, Name = Value) :-
    task_output(Task, Name),
    data_operation(retrieve, Name, Value).
retrieve_output(Task, Name) :-
    task_output(Task, Name).

%
% Inference execution
%

exec_inference(Inference, Input, Output) :-
    maplist(map(input), Input, In),
    invoke_inference(Inference, In, Out),
    map(output, Output, Out).
```

```
map(input, Input, In) :-
    nonvar(Input),
    data_type(Input, _), !,
    data_operation(retrieve, Input, In).
map(output, Output, Out) :-
    nonvar(Output),
    data_type(Output, _), !,
    data_operation(store, Output, Out).
map(_, Arg, Arg).

%
% Control primitives
%

repeat(Body, until(Condition)) :-
    repeat,
    once(Body),
    once(Condition), !.

transfer_task(obtain, Attribute, Attribute = Value) :-
    writef('\n Please enter the value of %w: ', [Attribute]),
    read(Value),
    nl.

%
% Trace info
%

write_task(start, Task, Input) :-
    ( main:trace(task) ; main:trace(task, Task) ) ->
    (   writef('\nActivating task "%w" \n', [Task])
    ,   write_arg('input ', Input)
    ).
write_task(end, Task, Output) :-
    ( main:trace(task) ; main:trace(task, Task) ) ->
    (   writef('\nTask "%w" terminated \n', [Task])
    ,   write_arg(output, Output)
    ).

write_arg(Type, Arg) :-
    is_list(Arg), !,
    checklist(write_arg(Type), Arg).
write_arg(Type, Name = Value) :-
    writef('  %w: %w = %w\n', [Type, Name, Value]).
write_arg(Type, Name) :-
    data_operation(retrieve, Name, Value),
    writef('  %w: %w = %w\n', [Type, Name, Value]).
```

Working memory

```
:- module('task-working-memory.pl',
  [ data_operation/3
  , data_operation/4
  ]).

:- use_module(['task-declarations.pl']).
```

```
:- dynamic
   data_store/3.

%
% Working memory primitives
%

data_operation(create, set, SetName, InitialValue) :-
    retractall(data_store(set, SetName, _)),
    assert(data_store(set, SetName, InitialValue)).
data_operation(create, list, ListName, InitialValue) :-
    retractall(data_store(list, ListName, _)),
    assert(data_store(list, ListName, InitialValue)).
data_operation(create, element, ElName, InitialValue) :-
    retractall(data_store(element, ElName, _)),
    assert(data_store(element, ElName, [InitialValue])).

data_operation(Op, Name, Store) :-
    nonvar(Store),
    data_type(Store, _), !,
    operation(retrieve, Store, Value),
    operation(Op, Name, Value),
    ignore(write_operation(Op, Name, Value)).
data_operation(Op, Name, Value) :-
    operation(Op, Name, Value),
    ignore(write_operation(Op, Name, Value)).

%
% Operations on all types (set, list, element)
%    store
%    retrieve
%

operation(store, SetName, Arg) :-
    check_arg(Arg, Set),
    retract(data_store(set, SetName, _)),
    assert(data_store(set, SetName, Set)).
operation(store, ListName, Arg) :-
    check_arg(Arg, List),
    retract(data_store(list, ListName, _)),
    assert(data_store(list, ListName, List)).
operation(store, ElName, Element) :-
    retract(data_store(element, ElName, _)),
    assert(data_store(element, ElName, [Element])).
operation(retrieve, SetName, Set) :-
    data_store(set, SetName, Set), !.
operation(retrieve, ListName, List) :-
    data_store(list, ListName, List).
operation(retrieve, ElName, Element) :-
    data_store(element, ElName, [Element]).

% Set operations:
%
%     member (returns a member randomly)
%     select (idem)
```

```
%     add
%     subtract
%     empty

operation(member, SetName, Member) :-
    data_store(set, SetName, Set),
    rnd_member(Member, Set).
operation(select, SetName, Member) :-
    retract(data_store(set, SetName, Set)),
    rnd_member(Member, Set),
    select(Set, Member, NewSet),
    assert(data_store(set, SetName, NewSet)), !.
operation(add, SetName, Arg) :-
    check_arg(Arg, Additions),
    retract(data_store(set, SetName, Set)),
    union(Set, Additions, NewSet),
    assert(data_store(set, SetName, NewSet)), !.
operation(subtract, SetName, Arg) :-
    check_arg(Arg, Deletions),
    retract(data_store(set, SetName, Set)),
    subtract(Set, Deletions, NewSet),
    assert(data_store(set, SetName, NewSet)), !.
operation(empty, Store, Bool) :-
    data_operation(retrieve, Store, []) -> Bool = true ; Bool = false.

% List operations:
%
%    member   (from first to last)
%    select   (idem)
%    select random
%    append
%    delete
%    empty    (similar to set operation)

operation(member, ListName, Member) :-
    data_store(list, ListName, List),
    member(Member, List).
operation(select, ListName, Selection) :-
    retract(data_store(list, ListName, List)),
    select(List, Selection, NewList),
    assert(data_store(list, ListName, NewList)), !.
operation('select random', ListName, Member) :-
    retract(data_store(list, ListName, List)),
    rnd_member(Member, List),
    select(List, Member, NewList),
    assert(data_store(list, ListName, NewList)), !.
operation(append, ListName, Arg) :-
    check_arg(Arg, Additions),
    retract(data_store(list, ListName, List)),
    append(List, Additions, NewList),
    assert(data_store(list, ListName, NewList)), !.
operation(delete, ListName, Element) :-
    retract(data_store(list, ListName, List)),
    delete(Element, List, NewList),
    assert(data_store(list, ListName, NewList)), !.
```

```
check_arg(Arg, Arg) :-
    is_list(Arg), !.
check_arg(Arg, [Arg]) :- !.

% Trace output

write_operation(Op, Store, Value) :-
    main:trace(working_memory, Ops),
    member(Op, Ops)
    -> write_op(Op, Store, Value).

write_op(Op, Store, Value) :-
    (   (Op==select ; Op=='select random' ; Op==member ; Op==empty)
    ,   writef('\nWorking memory operation "%w" on "%w"\n', [Op, Store])
    ,   writef('  with result: "%w"\n', [Value])
    ;   writef('\nWorking memory operation "%w %w" on "%w"\n', [Op, Value, Store])
    ).
```

B.2.2 Application-specific modules

Task declarations

```
:- module('task-declarations.pl',
  [ task/1
  , task_input/2
  , task_output/2
  , control_term/2
  , control_structure/2
  , data_type/2
  ]).

:- discontiguous
    task/1,
    task_input/2,
    task_output/2,
    control_term/2,
    control_structure/2.

%
% Task knowledge
%
% task(        Task name).
% task_input(  Task name, Input name).
% task_output( Task name, Output name).
% control_term(Task name, Term name).
%
% control_structure(Task name, Procedure).

task(              'Abstract new evidence').
task_input(        'Abstract new evidence',    'new evidence').
task_output(       'Abstract new evidence',    'new findings').
control_term(      'Abstract new evidence',    'focus set').

control_structure('Abstract new evidence',
  ( data_operation(store, 'focus set', 'new evidence'),
    repeat((
      data_operation(select, 'focus set', Focus),
```

```
      exec_inference(abstract, [Focus], NewFinding),
      data_operation(add, 'focus set', NewFinding),
      data_operation(add, 'new findings', NewFinding)),
    until(data_operation(empty, 'focus set', true)))
  )).

task(            'Clarify finding').
task_input(      'Clarify finding',  'finding').
task_output(     'Clarify finding',  'new observation').

control_structure('Clarify finding',
  ( exec_inference(specify, ['finding'], Observable)
  , transfer_task(obtain, Observable, Observation)
  , data_operation(store, 'new observation', Observation)
  )).

data_type('new evidence',      set).
data_type('new findings',      set).
data_type('focus set',         set).
data_type('finding',           element).
data_type('new observation',   element).
```

B.3 Inference-level modules

B.3.1 Generic modules

Generic inference activation primititves

```
invoke_inference(Inference, Input, Output) :-
    (   once(inference_function(Inference, Input, Output))
    ,   ignore(write_trace(Inference, Input, Output))
    ;   Output = []
    ).

write_trace(Inference, Input, Output) :-
    (   main:trace(inference)
    ;   main:trace(inference, Inference)
    ) ->
    write_inf(Inference),
    write_mc(Inference, Input),
    write_mc(Inference, Output, output).

write_inf(Inf) :-
    inference(Inf, String),
    (  String == [] -> Name = Inf ; Name = String),
    writef('\nInvoking inference %w\n', [Name]).
write_mc(Inference, Input) :-
    forall( nth1(Num, Input, MC), write_mc(Inference, MC, input(Num))).
write_mc(Inference, Value, input(Num)) :-
    dynamic_role(Inference, input(Num), GenName, SpecName),
    domain_retrieval(find_type, GenName, DomainEntity),
    (   var(SpecName) -> Name = GenName ; Name = SpecName),
    writef('  input :  %w "%w" (a %w)\n', [Name, Value, DomainEntity]).
write_mc(Inference, Value, output) :-
    dynamic_role(Inference, output, GenName, SpecName),
    domain_retrieval(find_type, GenName, DomainEntity),
```

```
    (   var(SpecName) -> Name = GenName ; Name = SpecName),
    writef('  output:  %w "%w" (a %w)\n', [Name, Value, DomainEntity]).
```

B.3.2 Application-specific modules

Inference declarations

```
:- module('inference-declarations.pl',
  [ inference/2
  , dynamic_role/4
  , static_role/2
  ]).

:- discontiguous
    inference/2,
    dynamic_role/4,
    static_role/2.

% inference(Internal name, External name)
% dynamic_role(Inference,     Input/Output,        General name, Specialised name).
% static_role(Inference, , Inference knowledge).

inference(     abstract,  'Abstract').
dynamic_role(  abstract,  input(1),       finding,     'Specific finding').
dynamic_role(  abstract,  output,         finding,     'General finding').
static_role(   abstract,  relation(abstraction, finding, finding)).

inference(     specify,   'Specify').
dynamic_role(  specify,   input(1),  finding,     'Finding to be clarified').
dynamic_role(  specify,   output,    observable,  'Dependent observable').
static_role(   specify,   relation(specification, finding, finding)).
```

Inference functions

```
:- module('inference-functions.pl', [ invoke_inference/3]).

:- ensure_loaded( ['inference-activation.pl'] ).

:- use_module(
  [ 'inference-methods.pl'
  , 'inference-declarations.pl'
  , 'domain-access.pl'
  ]).

inference_function(abstract, [In], Out) :-
    domain_retrieval(find_all, abstraction, Rules),
    rule_interpreter(Rules, In, Out, forward, single_pass, find_one).

inference_function(specify, [In], Out) :-
    domain_retrieval(find_all, specification, Rules),
    rule_interpreter(Rules, In, Out, backward, multi_pass, find_one).
```

Inference methods

```
:- module('inference-methods.pl',
  [ rule_interpreter/6
```

```
  ]).

%
% Computational techniques useed by inferences
%

rule_interpreter(Rules, In, Out, forward, single_pass, find_one) :-
    member([Premise, Conclusion], Rules),
    consistent(In, Premise),
    Out = Conclusion.
rule_interpreter(Rules, In, Out, backward, multi_pass, find_one) :-
    member([Premise, Conclusion], Rules),
    consistent(In, Conclusion),
    operand_of(Premise, Op),
    (   rule_interpreter(Rules, Premise, Out, backward, multi_pass, find_one)
    ;   Out = Op
    ).

consistent(X = Y, X = Y).
consistent(X = Y, X > Z) :-
    Y > Z.
consistent(X = Y, X >= Z) :-
    Y >= Z.
consistent(X = Y, X < Z) :-
    Y < Z.
consistent(X = Y, X =< Z) :-
    Y =< Z.

operand_of(X = _, X).
operand_of(X > _, X).
operand_of(X >= _, X).
operand_of(X < _, X).
operand_of(X =< _, X).
```

B.4 Domain-level modules

B.4.1 Generic modules

Domain access This module defines the access functions that can be used by the inference functions to retrieve domain knowledge from the knowledge base. The module uses indices defined in the module *domain-index* to to map the inference-level names onto domain-specific queries.

```
:- module('domain-access.pl', [domain_retrieval/3]).

:- discontiguous
    concept/1, concept/2, set/2, structure/2, relation/3, property/3,
    semantics/2, instance/3, value/3, tuple/2.

:- dynamic
    concept/1, concept/2, set/2, structure/2, relation/3, property/3,
    semantics/2, instance/3, value/3, tuple/2.

:- multifile
    value/3, tuple/2.
```

```
:- use_module([ 'domain-index.pl' ]).

:- ensure_loaded( [ 'domain-data.pl', 'domain-theory.pl']).

%
% Domain-knowledge access primitives
%

domain_retrieval(find_one, InfRelation, Tuple) :-
    domain_index(relation, InfRelation, DomainKnowTypes),
    member(Type, DomainKnowTypes),
    get_relation(Type, Tuple), !.
domain_retrieval(find_one, InfObject, Obj) :-
    domain_index(entity, InfObject, DomainKnowTypes),
    member(Type, DomainKnowTypes),
    get_entity(Type, Obj), !.
domain_retrieval(find_one, InfExpr, Expr) :-
    domain_index(expression, InfExpr, DomainKnowTypes),
    member(Type, DomainKnowTypes),
    get_expression(Type, Expr), !.

domain_retrieval(find_all, InfRelation, Knowledge) :-
    domain_index(relation, InfRelation, DomainKnowTypes), !,
    findall(KnowElement,
      ( member(Type, DomainKnowTypes)
      , get_relation(Type, KnowElement)
      ),
      Knowledge).
domain_retrieval(find_all, InfObject, Knowledge) :-
    domain_index(entity, InfObject, DomainKnowTypes), !,
    findall(KnowElement,
      ( member(Type, DomainKnowTypes)
      , get_entity(Type, KnowElement)
      ),
      Knowledge).

domain_retrieval(find_type, structure(I1, I2), structure(D1, D2)) :-
    domain_retrieval(find_type, I1, D1),
    domain_retrieval(find_type, I2, D2).
domain_retrieval(find_type, structure(InfName), structure(DomainEntity)) :-
    domain_retrieval(find_type, InfName, DomainEntity).
domain_retrieval(find_type, set(InfName), set(DomainEntity)) :-
    domain_retrieval(find_type, InfName, DomainEntity).
domain_retrieval(find_type, list(InfName), list(DomainEntity)) :-
    domain_retrieval(find_type, InfName, DomainEntity).
domain_retrieval(find_type, InfName, one_of(Defs)) :-
    domain_index(def, InfName, Defs).
domain_retrieval(find_type, InfName, DomainEntity) :-
    domain_index(_, InfName, DomainEntity).

get_relation(relation(RelationName), Tuple) :-
    get_tuple(RelationName, Tuple).
get_relation(property(Prop, ObjectType), [Value, Object]) :-
    get_object(ObjectType, Object),
    get_value(Object, Prop, Value).
```

```
get_relation(property(Prop, relation(RelType)), [Value, Obj1, Obj2]) :-
    get_value(tuple(RelType, [Obj1, Obj2]), Prop, Value).

get_entity(instance(C), I) :-
    get_instance(C, I).
get_entity(concept(C), SubC) :-
    get_concept(C, SubC).
get_entity(value_set(Prop, Type), Values) :-
    property(Type, Prop, Values).
get_entity(A, A) :-
    atom(A).

get_expression(expression(ObjType), [Obj, Prop, Value]) :-
    get_object(ObjType, Obj),
    get_value(Obj, Prop, Value).

get_object(Type, Object) :-
    get_instance(Type, Object).
get_object(Type, Object) :-
    get_concept(Type, Object).

get_instance(C, I) :-
    instance(C, I, _).
get_concept(C1, C2) :-
    sub_concept(C2, C1).

sub_concept(C1, C2) :-
    concept(C1, Supers),
    memberchk(C2, Supers).
sub_concept(C1, C2) :-
    concept(C1, Supers),
    member(C, Supers),
    sub_concept(C, C2).

get_value(Object, Property, Value) :-
    value(Object, Property, Value).
get_value(Object, Property, Value) :-
    instance(_, Object, PropValues),
    memberchk(Property = Value, PropValues).

get_tuple(Relation, Tuple) :-
    tuple(Relation, Tuple).
get_tuple(Relation, [Arg1, Arg2]) :-
    semantics(relation(Relation), symmetric),
    tuple(Relation, [Arg2, Arg1]).
get_tuple(Relation, [Arg1, Arg2]) :-
    semantics(relation(Relation), transitive),
    tuple(Relation, [Arg1, Tmp]),
    get_tuple(Relation, [Tmp, Arg2]).
```

B.4.2 Application-specific modules

Domain index

```
:- module('domain-index.pl', [domain_index/3]).

% Inference -> domain mappings
```

```
%

domain_index(expression, finding,       [expr(patient_data)]).
domain_index(entity,     observable,    [property(patient_data)]).
domain_index(relation,   abstraction,   [relation(qual_abstraction),
                                         relation(definition)]).
domain_index(relation,   specification, [relation(qual_abstraction),
                                         relation(definition)]).
```

Domain theory

```
% Domain knowledge format
%
% concept(Concept name,     Supertyes)
% property(Concept,         Property name,          Valueset)
% relation(Relation name,   Type first argument,    Type second argument)
% tuple(Relation name,      [First argument,        Second argument])

concept(patient_data,       []).
concept(quantitative_data,  [patient_data]).
concept(qualitative_data,   [patient_data]).

property(quantitative_data, temperature,        numberrange(35.0, 42.0)).
property(quantitative_data, diastolic_pressure,  numberrange(0, 300)).
property(qualitative_data,  fever,              [present, absent]).
property(qualitative_data,  blood_pressure,     [normal, elevated]).
property(qualitative_data,  hypertension,       [present, absent]).

relation(qual_abstraction,  expr(quantitative_data), expr(qualitative_data)).
relation(definition,        expr(qualitative_data),  expr(qualitative_data)).

tuple(qual_abstraction, [temperature >= 38.0,      fever = present]).
tuple(qual_abstraction, [temperature < 38.0,       fever = absent]).
tuple(qual_abstraction, [diastolic_pressure >= 95, blood_pressure = elevated]).
tuple(qual_abstraction, [diastolic_pressure < 95,  blood_pressure = normal]).
tuple(definition,       [blood_pressure =elevated, hypertension = present]).
tuple(definition,       [blood_pressure = normal,  hypertension = absent]).
```

B.5 Example output

```
3 ?-  abstract_example(_).

Activating task "Abstract new evidence"
  input : new evidence = [temperature = 40,diastolic_pressure = 100]

Invoking inference Abstract
  input :  specific_finding "diastolic_pressure = 100" (a [expr(patient_data)])
  output:  general_finding "blood_pressure = elevated" (a [expr(patient_data)])

Invoking inference Abstract
  input :  specific_finding "blood_pressure = elevated" (a [expr(patient_data)])
  output:  general_finding "hypertension = present" (a [expr(patient_data)])

Invoking inference Abstract
  input :  specific_finding "temperature = 40" (a [expr(patient_data)])
```

```
  output:  general_finding "fever = present" (a [expr(patient_data)])

Task "Abstract new evidence" terminated
  output: new findings = [blood_pressure = elevated,
                          hypertension = present,
                          fever = present]

Yes
4 ?- specify_example(_).

Activating task "Clarify finding"
  input : finding = hypertension = present

Invoking inference Specify
  input :  finding_to_be_clarified "hypertension = present" (a [expr(patient_data)])
  output:  dependent_observable "diastolic_pressure" (a [property(patient_data)])

 Please enter the value of diastolic_pressure: 100.

Task "Clarify finding" terminated
  output: new observation = diastolic_pressure = 100

Yes
```