ESPRIT Project P5248 KADS-II

An Advanced and Comprehensive Methodology
for Integrated KBS Development

Operationalising Knowledge Models

Cuno Duursma [*]
Walter Van de Velde [*]
(editors)
with extensive contributions from
Guus Schreiber [†]
Peter Terpstra [†]

[*] ***Vrije Universiteit Brussel***
[†] ***Universiteit van Amsterdam***

Document Id. KADS-II/T1.2/VUB/RR/004/1.0
CEC Reference Deliverable D1.2.3, Version 0.1
Date October 22, 1992
Distribution Consortium

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 Gemini Programator (S), Netherlands Energy Research Foundation ECN (NL), ENTEL SA (ESP), IBM France (F), Lloyd's Register (UK), Swedisch 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
75017, Paris
France
Phone: +33 1 40 54 66 32 ; Fax: +33 1 42 67 41 39
Contactperson: M. Georges
E-mail: mari@capsogeti.fr

Cap Gemini Programator
Norra Ringv. 36 Box 7
S-721 03 Västerås
Sweden
Phone: +46 21 14 44 50 ; Fax: +46 21 11 41 00
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

ENTEL SA
P. de la Castellana 141
28046 Madrid
Spain
Phone: +34 1 348 12 23 ; Fax: +34 1 279 10 74
Contactperson: S. Gala
E-mail: sgala@ic.eritel.es

IBM France
Rue de la vieille Poste
BP 1021, 34006 Montpellier
France
Phone: +33 67 34 40 43; Fax: +33 67 34 48 96
Contact person: G.Rondot
E-mail: rondot@mopprofs.vnet.ibm.com

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

Swedish Institute for Computer Science
KBS Department
Isafjordsgatan, 16428 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 3361 ; Fax: 49 89 636 441 50
Contactperson: C. Bauer
E-mail: cb@ztivax.uucp

Touche Ross Management Consultants
The Knowledge Based Systems Centre
Peterborough Court, 133 Fleet Street
London EC4A 2TR, United Kingdom
Phone: +44 71 936 3000 ; Fax: +44 71 583 1198
Contactperson: L. Land
E-mail:

University of Amsterdam
Department of Social Science Informatics
Roetersstraat 15, NL-1018 WB Amsterdam
The Netherlands
Tel: +31 20 525 6789; Fax: +31 20 525 6896
Contactperson: B.J. Wielinga
Electronic mail: wielinga@swi.psy.uva.nl

Free University of Brussels
Artificial Intelligence Laboratory
Office: Lentestraat 50-54 Mail: Pleinlaan 2, B-1050 Brussels
Belgium
Phone: +32 2 640 2535 ; Fax: +32 2 640 6326
Contactperson: W. Van de Velde
E-mail: walter@arti.vub.ac.be

This document was produced using the latex wordprocessor

ESPRIT Project P5248 KADS-II

An Advanced and Comprehensive Methodology
for Integrated KBS Development

Operationalising Knowledge Models

Cuno Duursma [*]
Walter Van de Velde [*]
(editors)
with extensive contributions from
Guus Schreiber [†]
Peter Terpstra [†]

[*] *Vrije Universiteit Brussel*
[†] *Universiteit van Amsterdam*

Synopsis

This document summarizes the results of Task I.2.3 "Design decisions for the operationalisation of knowledge models". The presentation of the *Common*KADS approach to design is in line with the basic *Common*KADS principle of risk-driven model development. We describe the *Common*KADS design model and then outline the design process from a risk-driven perspective. This is the general framework for KBS design. In addition we present contributions that apply the approach in the context of current practices and issues in software engineering. First, in order to link with conventional terminology, the relation between the modelling view and the "design from requirements" view is clarified. Next a KADS-specific structured design approach, called structure preserving design, is presented. Finally as a contribution to formal design methods, the usage of FML as a formal specification language is investigated. This document also presents realistic examples of how *Common*Kads KBS design works in practice. In particular a case study is presented that shows the whole process of modelling from problem identification to implementation.

Document Id. KADS-II/T1.2/VUB/RR/004/1.0
CEC Reference Deliverable D1.2.3, Version 0.1
Date October 22, 1992
Distribution Consortium

Preface

This document summarizes the results of Task I.2.3 "Design decisions for the operationalisation of knowledge models". It addresses one of the major gaps 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, as is also shown in a chapter of this document, working only 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 original 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.

In this document we present realistic examples of how *Common*Kads KBS design works in practice. Nevertheless these remain examples and our results should be validated by real applications. Consequently the *Common*Kads approach to KBS design will certainly evolve and it will be elaborated in various aspects. For example the model states, quality criteria and the integration with conventional methodologies for the moment remain at too shallow a level. These and other aspects of the design problem will be important subjects of study in the rest of the project and, no doubt, still many years after that.

Contents

Chapter 1

Introduction

Walter Van de Velde

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.

The purpose of this document is to present 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].

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? Basically our answer to such questions is contained in the following points:

---

[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.

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.

In this introduction a *Common*Kads approach toward KBS design is put forward out, which is based on these points. The following section briefly reviews several issues in KBS design. Then we investigate how software design fits in the model oriented *Common*Kads approach. Then we develop the outlines of our research approach and indicate where its contributions are most significant.

1.1 KBS Design

How is software engineering for knowledge based systems different from traditional software engineering? One strategy is to identify in what ways AI software is extreme. In this way we can avoid to the trap of simplifying the problem to apply existing solutions. 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 used to solve concrete problems directly and by users active in a domain of activity, i.e. without requiring a plan or procedure to apply them (as opposed to, for example, a statistics software package). This means that they must be grounded in real-world domains and highly integrated in the day to day activity of 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, 1990a]. 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.

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.

1.2 Scope of Design in *Common*Kads

The word "design" itself is used in various meanings. In its broadest sense it covers the process of transforming an "informal idea" into a finished design that can be readily implemented. In this meaning it involves much more than just designing 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.

In *Common*Kads the broad meaning of design is clearly too general. One of the powerful ideas is exactly the separation between the conceptual level and the software. Lumping all this together again in one "design" would be to throw away the child with the bath water.

> *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 model, the task model, the expertise model, the agent model and the communication model.*

This statement also captures our second scoping decision, namely to restrict to the design of the software. In its purest view the design model is the 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 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.

1.3 Modelling and Design

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

> *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 1.1 (taken from [Wielinga et al., 1992b]) summarizes the suite of models involved in the *Common*Kads methodology (see [deHoog et al., 1992] for a comprehensive description of the models and the dependencies between them). The design model describes the computational realization of the expertise and computational model within the boundary conditions specified by organization, task and agent model. More precisely it is a model of a (system) behavior in computational and representational terms.

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.[2] Those specializations reflect paradigms for design. They can be distinguished according to principle 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.

---

[2] 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. We will ignore this potential discrepancy.

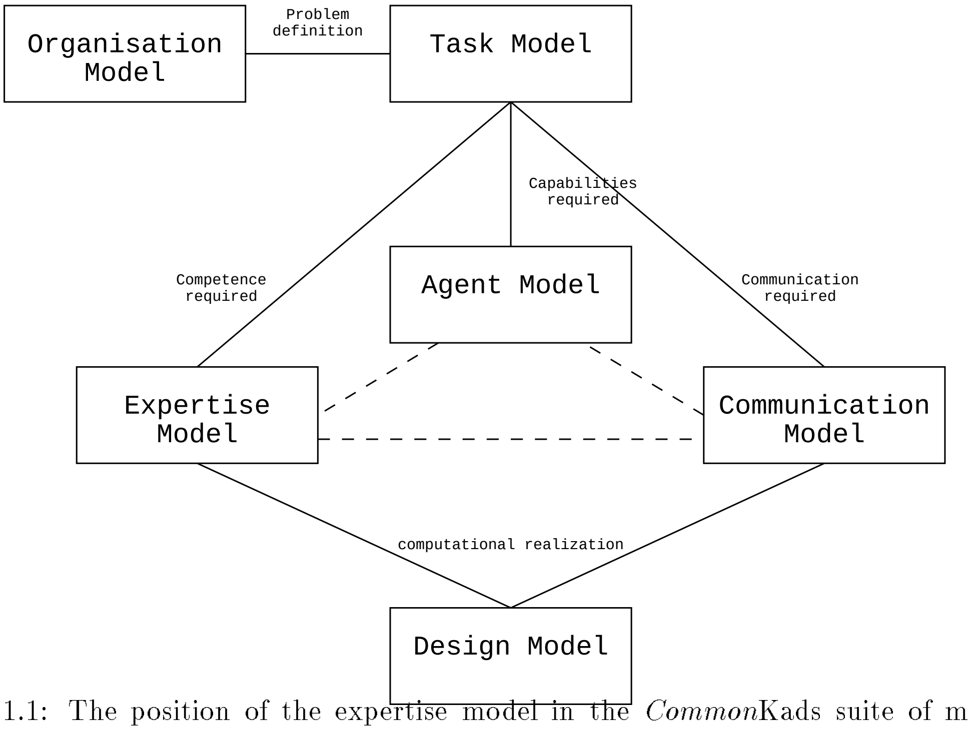


Figure 1.1: The position of the expertise model in the *Common*Kads suite of models. The organization model is the tool to analyse an organization. The task model captures the global task within the organization that deals with a certain function that needs support, including the assignment of tasks 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.

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.

1.4 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., 1989]:

- 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, it adds some additional insights and makes different emphasis:

1. For KBS design we are not developing new design methods. Rather we focused on what needs to be designed, leaving the design process as such as a second order issue.
2. The input to a traditional design process is a requirements specification. In *Common*Kads it had to be investigated where requirements come from, in particular what is their relation to the other models.

3. A risk driven process can be applied to design. The KBS specific aspects of design become clear in the KBS specific situations that may imply certain risks.

In order to reach concrete results we have focussed on the following:

- Elaboration of the *Common*Kads design model as a framework for design description
- Overview of requirements specification in the *Common*Kads modelling approach
- Selection of architecture as a crucial choice in design.
- Structure preserving design as a convenient way of operationalisation
- The role of formal specification in *Common*Kads design

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.

1.5 Overview of this document

This document contains the following chapters:

- foo bar

The notion of requirements is deeply rooted in the software engineering. It is rooted in the input-output view of the design process: start from a well defined requirements definition and produce a software definition. This ideal view seldom fits reality. What happens mostly is that requirements and software develop in a parallel way until they form a consistent set. In this way requirements are an odd thing in the *Common*Kads view. In the modelling approach different views on the product are being developed and documented in the *Common*Kads product model. The requirements are spread out over all the different views and the idea of requirements as input and the software as output is diluted. The process of modelling, and of designing in particular is one of finding a consistent product model. Which parts of this model serve as requirements or as solutions to these requirements is an artificial distinction.

Chapter 2

The Design Model

Walter Van de Velde

This chapter presents the *Common*Kads design model. The *Common*Kads methodology is model oriented. A KBS project involves the development of a series of models of an organization. Such a product model can, for instance, capture the KBS solution to a problem in an organization [deHoog et al., 1992]. A knowledge based system is typically part of such a solution and the design model describes its design in computational and representational terms. This chapter is a summary of the *Common*Kads design approach. It describes and justifies the role of the design model and touches upon the design process. A shorter version of this description is to be found as the design model entry in the *Common*Kads model set document [deHoog et al., 1992].

2.1 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 identify and anticipate 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 3.6. 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.[1]

---

[1] 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 *Common*Kads 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 may serve 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 *Common*Kads 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 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 reusable 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 can be 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.

Surely there are many more roles that one can think of. Maybe the most important but pragmatic reason for having an explicit design component in the *Common*Kads product

---

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

model is that it provides for a way to link knowledge based systems design to conventional design methodologies: no matter how one proceeds in design, the result is a software design that is contained in the design model. Care has been taken that the *Common*Kads design model indeed accommodates the more traditional models of a software design.

2.2 Contents of the design model

The design model records what are the important decisions that are to be made when designing an application. A decomposition of the design model is shown in Figure 2.1. We distinguish between three classes of decisions, namely related to *application design*, to *architecture design* and to *system design*.

- 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 system 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. 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.

We introduce an architecture design as a mediator between application design and system 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 system 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 course grained computational building blocks. Examples of architectures are blackboard systems or task specific architectures. Typically such 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 an implementation 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).[2]

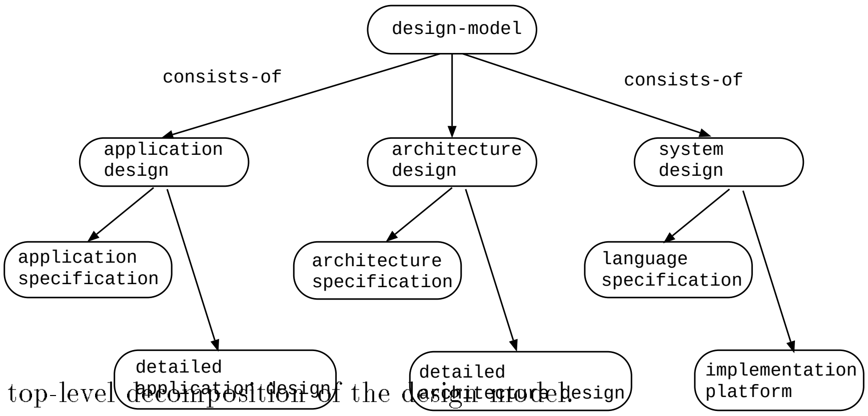


Figure 2.1: The top-level decomposition of the design model.

2.2.1 Application design

The application design consists of two parts: *application specification* and *detailed application design*. It is illustrated in figure 2.2.

Application specification The application specification consists of a high-level decomposition of the artifact in *sub-systems*. A sub-system can be a knowledge based system or an interaction sub-system. Other subsystems of conventional type (e.g., a data-base) can be present as well. There can be any number of subsystems of any of these three types ("KBS sub-system", "interaction sub-system" and "other sub-system"). The three types of sub-systems are introduced because they require different design paradigms supported by specific bodies of design knowledge. The design process for these three types of components is therefore conveniently treated as separate.

The different sub-systems in an application implement the functionality that is specified in the tasks within the task model or in the interactions in the communication model. [3]

The sub-systems are the starting points for further application design. Every subsystem has a *decomposition* into conceptual entities that realize the functionality. The goal is to come up with a consistent and sufficiently precise decomposition. This decomposition is typically hierarchical but need not be. The nature of the decomposition depends on the *decomposition paradigm* being used. Two well-known such paradigms from software engineering are function-oriented (or functional) decomposition and object-oriented decomposition.

---

[2] 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 envisaged but in practice this is not done.

[3] The high-level modularization of an application is sometimes called application architecture, not to be confused with computational architecture in the *Common*Kads design model.

Figure 2.2: The application design

For the KBS sub-systems the expertise model can be used as the decomposition. The paradigm is then "expertise". For a data-base on the other hand the conceptual schema (an entity relationship or NIAM diagram) could be used. For the interaction sub-systems the decomposition of transactions into transfer tasks which is provided by the communication model can be used. The usage of expertise and communication model for software specification is an instance of what is called *model-based specification* in software engineering. In *Common*Kads we argue that the expertise and communication models can be used as high-level design but we do not intend to say that high-level design and expertise or communication modeling are one and the same thing. See also the discussion in Chapter 3.

Detailed application design The application specification is realized by the detailed application design within the computational architecture of the application. It is the detailed specification of a "program" that can be run within that architecture. The detailed specification will typically use a program definition language (PDL) that is derived from the architecture specification in the architecture design model. When paradigmatic AI architectures - such as blackboards - are being used then AI programming techniques provide support knowledge for the detailed application 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.

2.2.2 Architecture design

The architecture design model consists of two parts: *architecture specification* and *detailed 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 is shown in Figure 2.3. The architecture specification specifies [4]

Figure 2.3: Description of the high-level architecture design component

The specification of a computational architecture requires a computational ontology and an inference cycle (abstract machine) on that ontology. In the following chapter we present in more detail techniques for architecture specification.

The detailed architecture design realizes the architecture within the target language. This is based on *computational techniques* for implementing pattern matching, various forms of chaining, agendas and so forth. The detailed design of a sub-system in the application instantiates the computational objects and calls the inference cycle.

The specification of an interaction architecture specifies an ontology of media and activities. Examples of media are windows, multi and hyper-media, ascii line printing. Examples of activities are printing, clicking a button, and so forth.

---

[4] It may be useful to include the top-level modularization of the application into sub-systems (i.e., the application architecture) in the notion of architecture.

The detailed design of the interface architecture realizes this functionality within a target (specification) language (for example MOTIF or X11) that is being decided upon in the system 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 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.

2.2.3 System design

In system design the lower level decisions about target language and hardware and software platform are documented. These can be different for any of the sub-systems in the application. These choices are often imposed by external requirements, availability, compatibility, prize, and so forth.

The main components of system design are:

Hardware configuration A description of the hardware configuration referring to types of machines, connections, integration devices and so forth.

Software configuration A description of the (commercial) software that is needed, its configuration and versions.

Environment A description of the work environment in which the system is to be installed, referring to elements of the organizational model. Environment includes location and user(s).

2.3 Conclusion

In this chapter we have presented and motivated the *Common*Kads design model.

The *Common*Kads model set document provides a number of guidelines and principles that determine what pieces of information should end up in a model. The ultimate justification of model contents, however, should be in its value for driving the development process. This is the second role of the models that was pointed out previously. Briefly stated, an element of a model is relevant if some operational characteristic of it can be shown to have an impact on a quality characteristic of the end product. To justify every part of the model in this way is a matter of validation. We will have only done this for the higher level model components. We do present some ideas for a more systematic approach toward this issue in the following chapter.

The design model is also illustrated in some depth in Chapter 8.

Acknowledgements The members of KADS-II Model Coordination Committee have greatly contributed with many comments to the development of the design model [deHoog et al., 1992].

Chapter 3

The Design Process

Cuno Duursma, Walter Van de Velde

This chapter describes the *Common*KADS design process. It does not present a design recipe, however. Our approach has become a mixture of guidelines and design principles, a general decomposition of the design problem (the design model) and a high level generally applicable design (the expertise model) that comes together with some transformation tools and support knowledge for making choices. Still with all this, design remains essentially a creative process.

In *Common*KADS the design process is, of course, risk-driven [Taylor et al., 1992]. A software design, represented in the *Common*KADS design model, is elaborated in anticipation of identified risks, where "risk" means the combination of threat and impact on a particular quality characteristic of the *Common*KADS product.

We will discuss these topics in the following order. After mentioning some general issues in design a high level design task decomposition is presented. For these tasks the support knowledge for the various design decisions is treated next. The final section deals with quality characteristics, risk-management and states of the design model. More detailed guidelines for specific design approaches are found in subsequent chapters.

3.1 The Nature of the Design Process

Why is design complex? In a nutshell the problem is in the difficulty of coordinating different decompositions: decomposition of the requirements, decomposition for analysis into high-level design elements, and decomposition for implementation into computational elements. Mapping from one decomposition into another and tracing back influences throughout these mappings is an extremely difficult constraint satisfaction task.

If designers want to perform any decomposition in any thinkable dimension of their problem they need to have an idea what their building blocks are. Some of our decomposition principles seem so 'natural' that they seem completely independent of any system design. This is however an illusion, but usually not a problem, because today's architectures require these decompositions. In biological systems, ecosystem and economic systems different decompositions are found.

The most common decomposition principles are functional and object oriented. Object oriented analysis and design has started its career from GUI (Graphical User Interface) implementation, and after several years it has become a new method of analysis. Other decomposition principles can be to decompose control, or dependencies, or location, or time.

A system analyst who knows the target techniques will analyze the problem in the domain already in an abstracted form of those techniques. When such an analysis has been made, the decompositions in the requirements can be preserved in the design. This is what has been named 'structure preserving design' and Chapter 5 is an example of it. It is however incorrect to think that one can analyze and decomposing the requirements of a system without any knowledge of the decomposition principle of the techniques and then expect a smooth structure preserving design process.

A second issue is that requirements are usually conflicting; and it is the designer's job to find a way out of this web; programs should always be fast but require little memory, require little knowledge but be accurate, flexible but simple, small but powerful. Designing is compromising between such requirements. Compromising makes the design less clear ("How on earth could this car be designed like that") because the reasons, i.e. the competing requirements and the decision the designer made to solve it, are usually not represented in the final system. The only help we can give here is to document the design process and link the documentation to the final code. This is in line with the *Common*Kads philosophy; develop, update, relate and keep thou models.

A third issue is the conflict between a binary requirement - piece of design mapping and general applicable parts of the design. If there is a one to one mapping between a piece of design and a requirement then it is easy to find out where to modify the design when the requirement has not been met. But in computer science we also like to develop general applicable parts. The advantages are well known, less memory, faster, re-usability, compact and understandable code. Another result is however that the one -to- one mapping disappears. A piece of design satisfies several requirements (of similar nature) in different parts of the system. Once a requirement has been violated it is still possible to identify the weak spot in the design, but changing the design can introduce conflicts with a set of other requirements on the system. We can only solve this by a typical design solution; to compromise.

3.2 The Design Model as Task Decomposition

The *Common*KADS design model as presented in the previous chapter can be considered as a high-level task decomposition. This interpretation is made explicit in Figure 3.1. A task decomposition simply groups things to be done in meaningful chunks (tasks). It implies no timing or logical dependencies between these tasks. These dependencies are, however, crucial to effectively control the design process.

3.2.1 Time Dependencies

The definition of the high-level components of the design model implies timing constraints among the design sub-tasks:

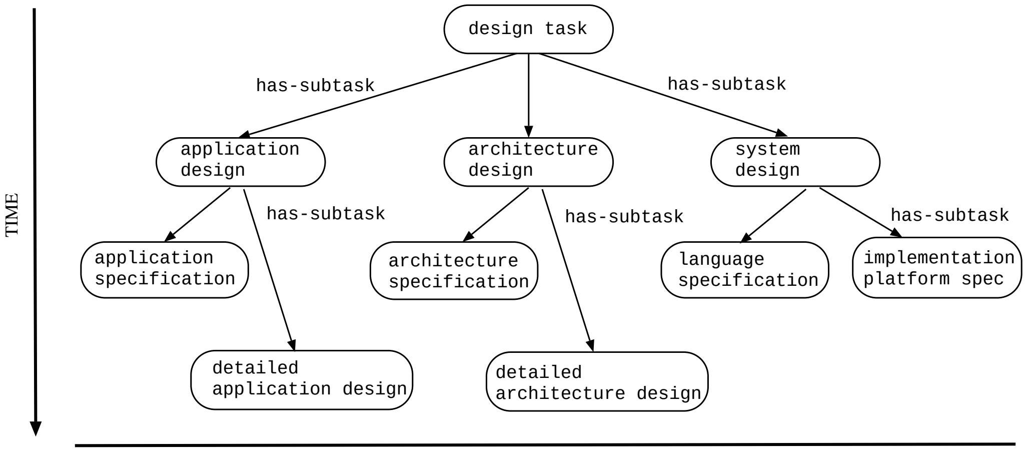


Figure 3.1: The design model interpreted as high-level decomposition of the design task. Task that are in horizontal rows can be done in parallel. Vertical arrangement represents an implicit time ordering.

- *Specification tasks can be done in parallel.* There is indeed no necessary dependency between the application specification, the architecture specification and the language specification. Of course this is an idealization. The choice of application analysis and of architectural paradigm will determine the ease of realizing the first one within the latter by subsequent detailed application design. Similarly the choice of architectural paradigm and implementation language will have an impact on the ease and quality of subsequent implementation (via detailed architecture design).

- *Detailed design of application can start as soon as the application specification and the architecture specification are finished.* The role of the detailed application design task is exactly to realize the application specification within the computational infrastructure that is specified in the architecture design. *Similarly the detailed architecture design can start as soon as the architecture specification and the language specification have delivered.*

- The above dependencies and timing implications of course only hold for interrelated parts of the design. For example, even with a partial application design and a specified architecture detailed application design can start. It seems, however, that *architecture specification has to be completed before detailed application can start. Similarly the language specification must be completed before detailed architecture design can start.*

A more or less standard top-down design process would be roughly as follows:

1. Select high-level application design paradigm

2. Elaborate high-level design

3. Select architectural paradigm

4. Elaborate architecture specification
5. Select platform and target language
6. Detailed design of architecture specification within target language
7. Detailed design of application within the architecture specification
8. implementation of architecture within target language and platform
9. implementation of detailed application design within implemented architecture

3.2.2 Logical Dependencies and Support Knowledge

The logical dependencies between the several design tasks are illustrated in Figure 3.2.

In conventional terms design starts off with a set of *requirements*, typically collected in a requirements definition document and an idea of the applicable techniques. Requirements are gathered from the various models in the *Common*Kads model suite. The major input for the design process in *Common*KADS is the expertise model which can be viewed as a specification of the problem solving requirements. Other inputs are user interaction requirements (i.e. the model of cooperation) and also a set of external requirements, such as costs, software and hardware. The relation between models and requirements is discussed in Chapter **??**.

In the end the designer should present a *software definition*, a detailed description of the software that needs to be produced. A software definition must be of such a level of detail that it can be picked up by a team of programmers who donot know the requirements and to be turned into a working system that actually satisfies the requirements.

The *Common*KADS design model brings together the various decisions that must be taken to elaborate the requirements definition. For each of these decisions there are paradigmatic options and support knowledge. These are also illustrated in Figure 3.2.

- The major paradigms for application design are functional and object oriented. Here a paradigm is characterized by the *decomposition principle* that is employed when describing a sub-system. Within a paradigm the sub-systems are decomposed in application specific components.
- Architectural paradigms provide one with a computational ontology and an abstract machine over this ontology. For example an object-oriented architecture prescribes objects, slots, methods and messages as computational ontology and a model of computation based on message passing. Note that this ontology is purely computational and therefore application independent.
- Paradigms of target languages are, for example, the well known procedural or declarative languages. Specific languages are LISP, Prolog and OPS5. The programmer is typically supported by a dedicated environment that provides editors, tracers, debuggers and so forth (KEE, ART, Quintus Prolog, NEXPERT and so forth).

The process of design, then, is to select one or several paradigms for application architecture, computational architecture and implementation language, and to specify a concrete application design, an architecture design and a concrete implementation language

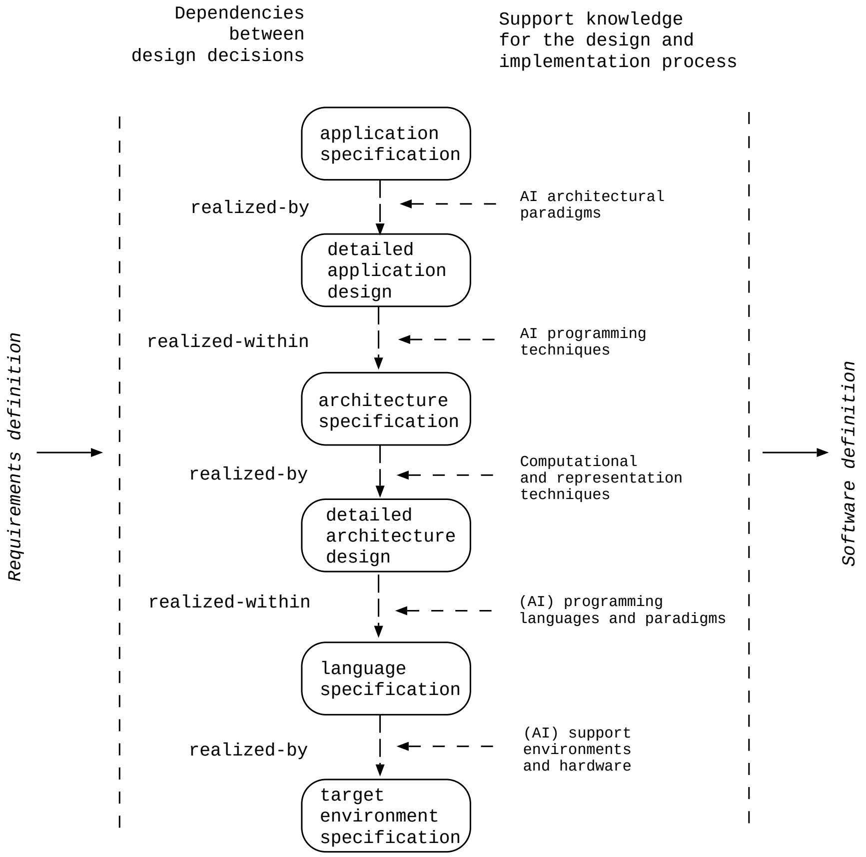


Figure 3.2: The dependencies between design decisions and the paradigms and background knowledge to support them.

within these paradigms. Finally the architecture design is realized within the implementation language and the application design within the architecture. These realizations are the detailed architecture design and the detailed application design, respectively.

- The *detailed application design* realizes the application design within the computational architecture. This is based on *(AI)-programming techniques* (rule-based programming, object-oriented programming, continuation programming, message passing, working with streams, and so forth). Such programming techniques are ways to use a computational infrastructure to realize complex representations or, for example state-space search or constraint satisfaction.
- The *detailed architecture design* realizes the architecture within the target language. This is based on *computational techniques* for implementing pattern matching, various forms of chaining, agendas and so forth. Many recent textbooks in AI-programming treat these techniques in considerable detail [Abelson and Sussman, 1985, Keene, 1989, Eisenberg et al., 1990, Bratko, 1986, OKeefe, 1990]

Detailed design relies on computational techniques and AI-programming techniques. Given an architecture, the designer will have to decide which computational techniques she is going to employ in the artifact. The nature of these decisions can be illustrated with an example concerning the skeletal architecture used in the early days of expert systems. This architecture was a very simple one: the two main components of a KBS were in this view an “inference engine” and a “knowledge base”. It is in fact a control-oriented architecture: the leading principle for the decomposition is the control relation between a declarative component (the knowledge base) and a procedural component (the inference engine) of the system. The computational design decisions that have to be taken in this architecture are concerned with:

- the representation formalism for the knowledge base,
- the algorithm for interpreting the knowledge base, and
- the control regime for sequencing interpretations of the knowledge base.

Typically, one type of technique was selected for realizing this architecture: e.g. a production system technique (e.g. production rules, rule interpreter with forward chaining, some form of conflict resolution) or an automated deduction technique (e.g. predicate calculus representation and a theorem prover).

One can view production systems and automated deduction as *computational paradigms*: they point to a group of related techniques and representations. Other well-known computational paradigms are *state-space search*, *parsing*, *constraint satisfaction* and *structure matching*.

Employing techniques that belong to only one particular paradigm simplifies the process of KBS design. In general however, one would not like to limit the designer to use one particular group of techniques. Some techniques are better suited for solving certain (sub-)problems than others.

The detailed planning of the actual design activity is risk-driven (see section 3.6). It is therefore important to understand what aspects of the design are particularly important in which situations and what their impact will be on the final quality characteristics of the product. What follows is an overview of these aspects for the three top-level design components: application design, architecture design and system design.

3.3 Application Design

3.3.1 Paradigmatic Specification Techniques

There exist at least three approaches to the specification of software: informal specifications, algebraic specifications, and model-based specifications.

- Informal specifications are based on natural language descriptions. These have the advantage of being easily read, but often suffer from ambiguity and imprecise meaning.

- Formal specification are based on statements that have a precise semantics and thus use some form of mathematics. Formal specification techniques, though they are harder to use, have a number of advantages [Sommerville, 1989], among others:

    1. it may be possible to prove that an architecture satisfies certain properties or eliminates a certain class of errors;

2. formal specifications may be automatically processed;
3. formal specifications may be animated to become a prototype system;

In addition formal specification of architectures allows one to compare different architectures and to understand their respective advantages. In practice formal specifications are combined with informal specifications in natural language.

Within formal specification techniques one distinguishes between the algebraic and the model-based techniques.

- An algebraic specification describes computational entities in terms of the operations that act over it, and the necessary relationships between these operations. Entities are typically sorted and the operations described by their signature and a set of axioms. Operations are either constructor operations or inspection operations. Algebraic specifications are simple and concise for simple operations but modeling the dynamic aspects of control and processes may be difficult. Special algebraic formalisms have been developed for this purpose (e.g., process algebras [en W.P. Weijland, 1990]).
- Model based techniques construct a model of a system in terms of mathematical entities like sets and functions. The model corresponds to the state of the system which is changed by the action of the functions on it. Well known examples of a model-based specification technique are petri-nets [Brauer et al., 1987], Z [Hayes, 1987] or VDM [Jones, 1986a].

3.3.1.1 Design from non-formal requirement definition

In software engineering the requirement specifications are usually decomposed in either a functional (*functional design*) or in an object oriented (*object oriented design*) way. The distinction between object oriented and functional approaches can have already impact on the requirement specification stage. For instance algebraic specifications map more directly on object oriented approaches than on functional approaches.

3.3.1.2 Design from formal requirement definition

A different approach can be taken if the specifications are completely formalized by algebraic or by a combination of pre- and postcondition of functions. A design can then be made by truth preserving transformations to a selected implementation language.

Algebraic specifications model usually objects together with their operations. Usually also some inheritance by importing and exporting classes of objects is available. In *Common*Kads the algebraic specifications in ML^2 (cf. [van Harmelen and Balder, 1992]) have an ontology which is similar to the Expertise Model. Therefore the objects around which the algebraic theories are grouped are not data objects but concepts of the expertise model. Algebraic specifications in terms of ML^2 are object oriented, but the objects are functional entities!

Analysis	**Requirements**	**Design process**	**DDL**	**Implementation**
OO analysis	Algebraic spec	Object Oriented design	Object Oriented design language	Object Oriented implementation language
	Model based spec (VDM, Z)	Animation, truth preserving manual and automatic transformations		
Functional analysis	Ada based specifications (SADT, PSL, PSA)	Functional design	Ada /PDL	Ada
Dependency analysis	Constraint specifications			Constraint satisfaction
*Common*Kads analysis	Task, agent, organizational, and communication model	Conceptual Expertise modelling, then structure preserving design (functional design)	First CML, then any DDL	Any platform
idem	idem	Formal expertise modelling	(semi) automatic transformations	Prolog

3.3.2 Software Engineering & *Common*Kads

In the field of Knowledge based systems usually other virtual machines are built on top of general purpose implementation languages. Examples of these are rule based system shells, GBB, Case based reasoning shells, knowledge representation shells, truth maintenance systems and constraint satisfaction systems. The availability of such an architecture makes it attractive to design the system in such a way that the detailed design maps directly on the chosen virtual machine.

Like in conventional software engineering the approach has been to move up implementation languages to the design and requirement specification. After the invention of rule based languages it was thought that these implementation languages were sufficiently close to a human way of thinking that they could be directly applied for both design and requirement specification. The introduction of Prolog had similar effects.

Methodologies for expert system engineering are different in this respect because they try to abstract from a particular implementation language. The advantage of this is that the requirements definition can be done in a formalism that is fully adapted to human approaches to problem solving. Disadvantages are the larger step from requirements definition to design definition, and the exact meaning of requirements written in terms of models provided by the methodology.

The *Common*Kads methodology is meant to be independent of the target DDL and the target implementation language. The approach is based on an initial task decomposition and an analysis in terms of either inference structures, domain models and problem solving methods. The task decomposition can best be compared with a functional approach. The modelling in terms of domain models can best be compared with an object oriented approach. The inference structures and problem solving methods are always associated with tasks; therefore they seem also to be part of the functional approach [1]

3.3.3 Abstract Software Design: Expertise Model

The *Common*Kads expertise model can be compared to a high level software specification (technical design) of conventional software engineering. It is part of the design process, which is based on the requirements definition.

The expertise model is usually applied to make a model of the problem solving competence of the problem solver, those tasks in the task model which are assigned to the expert-system component of the system required. The language and structure of the expertise model is best suited to both support the analysis of the expertise of the domain expert, and also to provide a high level design of the desired use of expertise in the required system.

In Newell's terms, the computational techniques describe the symbol-level realization of the agent. It is important to realize that there is a trade-off between (i) making the knowledge-level model more specific, i.e. by introducing more refined elements, and (ii) a more elaborate specification during design. The decision whether to do the first or the latter depends on whether one needs to represent explicit control on certain operations in the model of expertise. The borderline between model of expertise and design is thus in a sense governed by the level of *granularity* that is required of the model of expertise. For example, in **OFFICE-PLAN** (a system for allocating offices to employees, see [Karbach et al., 1989]) the actual allocation inference is modeled as one knowledge source *assign* in the model of expertise and is realized in the actual system through a complex constraint satisfaction technique. If it would have been necessary to exercise control on this technique, then one would need to model constraint satisfaction "at the knowledge level".

3.4 Architecture Design

3.4.1 Architectural Paradigms

In the literature the term "architecture" is used in many different ways. We use the term to denote a description of the high-level computational infrastructure of the artifact to be developed and their inter-relations. The nature of the prime ingredients described in an architecture varies depending on the *architectural paradigm* that is being used. Such a 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

---

[1] This is a difference between the *ComMet* methodology and KADS-I; ComMet starts an object oriented analysis of the problem by building a system model in terms of domain models. KADS-I makes a functional task analysis and a logical process diagram between the tasks as an initial system model.

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, 1992a]), 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.[2] Many of the debates in software engineering have been about the "right" decomposition principle. Chapter 5 argues for an architectural paradigm that represents a combination of these three perspectives. This is in line with recent proposals in software engineering, such as advocated by [Rumbaugh et al., 1991].

3.4.2 Architecture Specification

The specification of the architecture entails two steps:

Choice of the architectural paradigm

The choice of the perspective that guides the decomposition in the architecture: e.g function-oriented, object-oriented. In real-life practice this paradigm choice is often not noted as an explicit design decision. It is often determined by the background and experience of the system designer or by software-engineering 'fashions".

Architecture specification

Given an architectural paradigm, the designer will have to use the analysis input (the conceptual model) to specify a suitable architecture. From a methodological

---

[2] This statement is not completely true for recent developments in the filed of so-called 'user-interface management systems" such the kernel of the KEW workbench [Anjewierden, 1991].

point, this activity can be supported by providing the system designer with *skeletal architectures*: prototypical decompositions that have to be instantiated for a particular application. In the following section we list different types of such skeletal architectures that could be useful for *Common*KADS designer.

The specification techniques that were mentioned in Section 3.3.1 can be used here as well. An additional technique which is sometimes used is the paradigm specific technique. Paradigm specific techniques for architecture specification are tailored toward a specific class of architectures. For example blackboard generators such as Erasmus [Baum et al., 1989] and BSG [Silverman et al., 1989] use a formalism which is specific to describing blackboard architectures in predefined terms. The advantage of this approach is that many design decisions which are specific to the paradigm can be left implicit. In addition special purpose system generators can be constructed that, from the architecture specification and application specific code, generate an application. On the other hand these formalisms are inherently restricted, and the specification of heterogeneous architectures that combine different paradigms may be difficult. For an overview of architecture specification techniques see [Velthuijsen, 1992] and [van Langevelde et al., 1992].

To specify an architecture in *Common*KADS a combination of algebraic and model based specification is proposed:

- 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 methods specify the effects of the basic mechanisms in the architecture that act over computational objects. Computational methods allow no external control.

  Methods are defined in a model oriented way with respect to the architecture.

- The computational architecture itself then is specified in terms of the computational objects and computational methods that constitute a system, the different computational states that a system can be in, and the control over the computational methods according to the states.

  The computational architecture defines the notion of a system as a structure of computational objects. It also defines notions of states for the system and computational methods are defined over this vocabulary, ultimately using the primitive operations (constructors and inspectors) over the computational objects. The architecture thus describes an organization of computational methods (processes) cooperating by communicating via the qualified states that the architecture may be in.

In addition to the specification of the computational objects one defines a program definition language (PDL) in which to describe the computational objects and calls to the computational methods. This PDL typically corresponds closely to the programming language that is being used for working within the architecture, i.e., for the detailed application design. It is required that each object's constructor, inspector and every method has a PDL (parser) specified. By defining the program definition language per object and method the specification of heterogeneous architectures is facilitated.

3.4.3 Architecture Selection

The computational architecture is the pivotal element between application design and implementation. The selection of an architecture is important since it will determine to a large extend the ease of implementation and maintainability of the system. The computational architecture provides the programmer with a level of abstraction that should be sufficiently close to the high-level design of the application.[3] The selection of a computational architecture must be based on the following considerations:

1. When is a certain paradigmatic architecture appropriate?
2. Which specific architecture within a paradigm is suited for a specific application?
3. How can the application specification be mapped onto the architecture?

In what follows we briefly describe four major types of architectures: task specific shells, task level architectures, KADS specific architectures and general architectures. One could add *executable specification languages* and environments to this list [ten Teije et al., 1991, van Harmelen and Balder, 1992]. However, these languages are often designed for validation purposes (to enable simulation of the designed behaviors, for example) and not for code generation.

Task-specific shells: Representations of problem solving behaviors are directly linked to ready-made complete execution shells. The task structure, problem solving method and domain knowledge structure is hard-coded and cannot be adapted by the system developer. Domain-specific knowledge is to be supplied by the developer to obtain an executable application. Task-specific shells are a response to the role-limiting methods approach in which model construction is viewed as the selection and instantiation (i.e. filling in domain-specific knowledge) of predefined models of problem solving. Well-known task-specific shells are MOLE [Eshelman, 1988a], SALT [Marcus and McDermott, 1989] and OPAL [Musen, 1989].

To add some flexibility to these task-specific shells, PROTEGE II enables system developers to combine mechanisms to construct problem solving methods and to define the role of domain-dependent knowledge [Puerta et al., 1991]. The output of PROTEGE II is a task-specific shell.

Task-specific architectures: Generic tasks are linked to special dedicated representation languages. Inference mechanisms and control within these high-level languages are fixed. Only domain-specific knowledge may be added. Operationalizing knowledge level models involves configuring the instantiated dedicated representation languages related to each generic task in the knowledge level model. Configuration is realized through message passing. GT supports the idea of task-specific programming languages [Bylander and Chandrasekaran, 1988].

Task-independent application kits: These kind of design environments support the white-box approach [Steels, 1992, Steels, 1990b]. Reusable representations at the

---

[3] The interaction architecture is dependent on the system requirements related to interaction with the user. We will treat it in considerably less detail.

knowledge level are offered. These representations are linked to predefined programming code. The application kit incorporates a compiler or interpreter to translate the code into objects at the execution level. This approach supports the task-specific programming language idea by supplying reusable representations (at the knowledge level and code level) and adds flexibility by allowing the system developer to inspect and modify the reusable representations. Constructs in the knowledge level analysis which have not a computational counterpart in the application kit can be created in the spirit of task-independent programming languages.

Task level architectures: these are very high-level representation languages which establish a close correspondence between primitive components at the knowledge level and computational constructs at the symbol level. Constructing an implemented system with task-independent programming languages involves mapping the knowledge level model onto the programming language with few design decisions left to the system developer. These programming languages offer a compiler or interpreter which may translate the programming into executable code. Examples of task level architectures are CML [Vanwelkenhuysen and Rademakers, 1990a], Model-K [Karbach et al., 1991] and OMOS [Linster, 1992].

General architectures: these offer high-level computational constructs that however have no intended interpretation in knowledge level terms such as task or model. The most general classes of these architectures are functional (e.g., [Chandrasekaran, 1987]), object-oriented (e.g., [Cointe, 1988, Mitchell et al., 1990]) and blackboard (e.g., [Velthuijsen, 1992]). Making a detailed application design in such an architecture requires a serious effort to map conceptual model elements onto computational constructs. These architectures require skill of the designer/programmer but are useful when hybrid application with complex control or representation needs are to be implemented [Laird et al., 1987]

These different approaches towards supporting a structure-preserving design of knowledge level models may be situated on a spectrum relating efforts required from a system developer to create an implemented system (with task-independent programming languages at one end) and the limited flexibility offered to adapt or fine-tune the design and/or implementation to the specific needs (with task-specific shells at this end). Task-independent application kits seem to offer a good trade-off between the two extremes.

Clearly the KADS specific task level architectures are the easiest when flexibility is required with a structure preserving design approach from a *Common*KADS expertise model.

3.5 System Design

At the lowest level the designer has to choose a software environment that is to be used for implementation. This choice is often restricted by very down-to-earth circumstances: availability, company policy, integration requirements, available hardware, compatibility, price, and so on and so forth. The choice is, however, important as it may clearly have an impact on the quality of the product, for example in terms of portability or efficiency.

With a software environment we mean some programming language (with possibly additional components, e.g. libraries) that is used as the basis for implementation: e.g. KEE [Fikes and Kehler, 1985], EMYCIN [van Melle et al., 1981], Prolog, LISP, etc. A software environment supplies the designer with a number of predefined (=implemented) implementational constructs. In [Schreiber et al., 1987] a distinction is made between *closed* and *open* environments.

In a closed environment the set of available techniques and representations is fixed and not expandable. An example of such an environment is EMYCIN. An open environment offers possibilities for expanding the set of methods and representations. Open environments can be further divided into *weak* or *strong* environments, depending on the size of the set of predefined techniques. Prolog can be viewed as an example of a weak environment. Its built-in facilities are very general-purpose: unsorted Horn clause logic, unification and backward chaining. KEE can be seen as an example of a strong environment. It provides higher-level primitives for implementing computational techniques, such as frame and production rule representations, hierarchical structuring of frames and rules, a classifier and various interpreters,

There is a clear dependency between the computational decisions on the one hand and the choice of an environment on the other hand. The techniques chosen can influence the choice of an environment and vice versa. Ideally, the chosen environment should offer primitives for realizing a large variety of techniques and thus limit the amount of required implementation effort. In a weak environment such as Prolog or LISP, it is necessary to build the techniques on top of the language.

It should also be noted that in real-life KBS development the choice of a software environment is often dictated by external requirements: costs, availability, etc. This environment then acts as a constraint on the overall design process: e.g., if, for some external reason, a system like EMYCIN has to be used, this implies that the designer has only limited freedom in making computational decisions or has to devise clever ways of implementing the desired constructs using the limited means that the environment offers. A typical example of this last phenomenon is the use of parameters to represent control implicitly in MYCIN [Clancey, 1983].

3.6 Risk Driven Design

The general guideline for design using the *Common*Kads methodology is to address high-risk matters first. Also in the other development processes in this methodology the risk driven approach has been put forward [Taylor, 1992]. Experienced designers will take the risks into account by comparing the project with previous successes (or failures). For less experiences designers it is useful to think explicitly about the risk of each part of the design. Thus, instead of providing an overall design process we think that the process is the result of a continuous evaluation and decision process according to the risk-driven spiral model. The result may be a straightforward top-down development process if the designer has both experience with the problem as well as with the techniques, but will typically be more complex.

It is not necessary to add much more to this then to provide the reader with a list of things that should be taken into account in such a risk assessment. The next section shows our attempt, a mixture of our design experience and common sense.

3.6.1 Quality, Risks and States

Apart from the high-level task decomposition of design *Common*KADS provides no ready-made plan to work through the design of a KBS. All the activities in *Common*KADS are risk-driven and so is the design process [Williams et al., 1992]. This means that models (among others) are developed in anticipation of identified risks. The process is therefore basically aimed at quality, i.e., at producing a high quality product through a high quality process. This process consists of pushing the product model through a number of states with predefined form and quality characteristics. The top-level user-relative quality characteristics in *Common*KADS have been adopted from ISO9126. They are [Williams et al., 1992]:

- functionality
- reliability
- usability
- efficiency
- maintainability
- portability

For each of these a *quality model* must be defined: a decomposition in more and more precise quality characteristics that can be applied either to the overall product, to intermediate products or product components, or to documents, and that must eventually be operational (measurable).

By defining landmark states of the product model the project can be steered such that the product model moves through these states toward the project's pre-specified quality objectives. Activities in the project are planned exactly to achieve this. In the following subsection we present ideas on how these states can be found.

3.6.2 States and Quality Characteristics

A state is specified in terms of a form and a quality characteristic. An exhaustive collection of states for the design model can be based on the componential definition of the design model. For each component it is meaningful to test for the presence of that component:

- SOFTWARE-PLATFORM-ESTABLISHED
- ARCHITECTURE-SPECIFICATION-ESTABLISHED
- and so forth

Most of these will not be very useful or meaningful. Meaningful states must have at least one quality characteristic specified. Our strategy is to apply the quality framework to each of the components. This means that for application design, architecture design and system design a quality model will be elaborated for each of the six quality characteristics. Certain of the quality characteristics do not make sense for all of the components. For example it is hard to imagine what the reliability of an application specification means in an end-user relative sense.

As a first step toward elaborating state definitions I suggest the following strategy for arriving at a meaningful set of states of the design model. The following steps must be taken:

1. For every high-level quality characteristic and every top-level component in the design model ask the question whether that component has an impact on the quality characteristic. For example "Does the architecture specification have an impact on the reliability of the product?".
2. If the answer is no then stop exploration of this characteristic. For example "Application specification does not have an impact on portability".
3. If the answer is yes, then ask "In what way? Specify an operational property that produces that impact". For example "The architecture specification must be such that no impasse can occur in the inference cycle".
4. Replace the characteristic with one from its model and ask the same questions. For example "Does the architecture specification have an impact on the maturity of the product" (No), "Does the architecture specification have an impact on the fault tolerance of the product" (Yes), and "Does the architecture specification have an impact on the recoverability of the product" (Yes). Go on until the answer is no longer reducible to a single lower-level characteristic.
5. A candidate landmark state can now be defined that links the lower level quality characteristic to the model form "component-established". For example a candidate landmark state becomes "Architecture specified such that no impasse can occur in the inference cycle".
6. Identify the sub-components to which the same quality characteristic applies. For example "Does the inference cycle have an impact on the reliability of the product? Yes: The inference cycle must be such that no impasse can occur in it". There can be more than one sub-component for which the answer is yes. In that case the state of the component becomes landmark and the lower component states become intermediate. Otherwise the state of the sub-component becomes candidate to be landmark.
7. At all times the characteristic can be reformulated to a more specific form applicable to the component. For example it was switched from "the inference cycle is reliable" to "the inference cycle has no impasse". This produces no states, just reformulates them in a more and more specific way.
8. At all times a characteristic may be further split. For example we can go on requiring "the inference cycle has no tie-impasse". This produces additional intermediate states.

The idea behind this strategy is to identify the closest possible correspondence between model components and product quality.[4] The next step is to find out about typical

---

[4] Note that the exercise could be done for any model in the *Common*KADS model suite

situations in which a certain risk (impact on quality) may occur. These elements together must eventually allow for a design process with build in quality management.

3.6.3 Design Guidelines: Identifying Risks

In this section we provide a list of risks that may affect the need for the design model to be developed in a certain state. This list is preliminary and meant to illustrate the kind of analysis that may be going on during design.[5]

1. Can you describe typical situations (risks) which might affect the need for the model (e.g. prioritise, de-emphasize?)
    - Multiple programmers involved (risk: inconsistent data model and redundant functionality): the design model should be developed to ensure a common data-model and avoid redundancy.
    - Low-level language is target language, conventional environment being used (risk: hacking): the design model must be developed especially in the computational architecture part that must lift the target language to a level of abstraction where programming the application becomes easier.
    - Maintenance is crucial (risk: maintenance fails): design model must be developed in great detail.
    - Implementation of artifact will be done by third party or client himself (risk: contractual quality of software specification not met): design model is crucial as product and must be developed in great detail.
    - Integration with existing software needed (risk: external interface mismatch): design model must be developed especially in the interface to the other software.
    - Speed and performance are important (risk: performance requirements not met): design model must be developed and carefully evaluated with respect to performance.
    - Functionality under-specified: this is a risk that the requirements are too general to map onto detailed functionalities. The design model must be elaborated in its application design.
    - Multi-user system: design model must be developed in great detail, especially in the data management.
    - Highly interactive system (graphical): design model must be developed in application design of interaction subsystems and the interaction architecture. Detailed design of dialogues and menus is crucial. (this is a special case of functionality under-specified risk, related to usability quality characteristic).
    - Critical application (risk: reliability too low): design model developed in great detail and checked through in relation to other models.

---

[5] This section is based on a questionnaire that was put together by Clive Bright (LR) to initiate the investigation of concrete states, risks and quality characteristics.

- Portability is important (risk: not portable unless redesigned): design model should be elaborated mostly in the application and architecture design. Especially architecture and its detailed design in a suitable program definition language are important.

2. In what circumstances could you avoid building the model? Would there be any consequences of omitting it in a project?

   - Task specific shell available: the design model can be de-emphasized since the shell supports the design by implementing role-limitations and providing role-representations that are suitable for the job.
   - Small project, programmer and knowledge engineer are the same person, no maintenance requirements: the design model can be omitted or de-emphasized. At least the system model should be developed (target language, platform). Highly unlikely.
   - Throw away software, simple application: don't use KADS
   - Omitting the design model is likely to lead to software that can not be verified, maintained and will contain bugs. Dependence on the original programmer remaining available.

3. Describe a basic set of steps/activities which you carry out to construct the model

   - Application design: the decomposition into subsystems (KBS, interaction, other)
   - Application specification: logical decomposition of subsystems up to a level which is implementable within architecture
   - Detailed application design: realize application specification within architecture PDL
   - Architecture specification: describe computational objects and computational mechanisms
   - Detailed Architecture design: describe implementation of architecture in target language
   - Choose target language: choose of specific target language(s)
   - Choose platform: hardware decisions

4. What products are produced along the development of a model (documents etc)?

   - Requirements document: input to the design process from other models
   - Software definition document: detailed software description, output of the design process
   - Application design document: specification of application
   - Architecture design document: specification of architecture
   - System requirements document (low level target language and platform).
   - Interface design: detailed description of interaction architecture and functionality

  - User guide: detailed documentation of system functionality

5. For each step state what structural criteria should be achieved (e.g. top-level task decomposed into sub-tasks).
6. Identify the quality characteristics that could/should be relevant at each step.
   - Application design: functionality specified in detail
   - Application design of interaction subsystem: assure usability
   - Architecture design: performance, portability, reliability, extensibility, maintainability, transparency.
   - System design: efficiency, cost effectiveness
7. List the components which comprise the model (e.g. Functional, structural, constituents of the organization model;...)
   - Application (sub-systems, specification, detailed), architecture (specification, detailed), target language, platform.
8. For each component of the model describe the risks that would lead you to build that component
9. For each component list at least one quality characteristic that will be affected by the production of that part of the model
10. For each component of the model describe a set of steps /activities which you carry out to construct model components. Please indicate steps which support validation or verification of the model
11. for each component of the model list the techniques which support its development (e.g. analyze current situation: interview, monitor,...).
    - For application specification: expertise and communication modeling, functional design or object-oriented design methodologies.
    - To detailed application design: AI programming techniques
    - To architecture specification: AI architectural paradigms knowledge used.
    - To detailed architecture design: computational and representational techniques
    - To language specification: knowledge about (AI) programming languages and paradigms
    - To system specification: knowledge about (AI) support environments and platforms
12. Describe the form(s) of notation used to describe a model. If different components of a model use different forms of notation then describe these.
    - The forms for high-level application design depend on the design paradigm being used. It can be the model of expertise, communication model, functional decomposition or object-oriented decomposition. Others are possible as well (entity-relationship).

- The forms for architecture design are abstract machine descriptions: data model and inference cycles.
- The format for detailed design is a program definition language that is derived from the target language (e.g. lambda calculus like) or from the architecture specification (e.g. an abstract rule format).
- The format for system design is standard system hardware and software configuration description.

Acknowledgments The quality discussion has been greatly influenced by the *Common*KADS Quality Viewpoint [Williams et al., 1992]. Clive Bright authored the questionnaire for the initial exploration of states, quality characteristics and risks.

Chapter 4

Requirements Identification

Cuno Duursma

The design process in software engineering takes a requirement definition and produces a software specification. In order to improve understanding of the *Common*Kads design approach we show how requirements can be retrieved from the model set. Figure 4.1 shows the analysis process of *Common*Kads in some detail. The five models of the 'desired situation' are instantiated in a process which can be compared to software engineering's requirements identification. Parts of the models can be combined in order to produce a requirements definition document. This requirements definition is one of the inputs of the design process. It is the purpose of this chapter to find out which requirements can be retrieved from what model. The role of the model of expertise is the most controversial one in this respect, because it has been a issue of debate whether or not the expertise model is or can be used as an analysis model, a requirement definition, or a high level design. In chapter 5 the expertise model is used as a high level design. In older Kads work it has was used as an analysis model [Breuker and Wielinga, 1985], and in section 4.1.1.4 we discuss the possibility to use it in a requirement definition.

Requirements should be consistent and complete. In the Model set this can be tested by inspecting the models and their relations. We show how completeness and consistency of the models can be tested and what kind of dependencies exist.

First we summarize the types of requirements documents produced in Software engineering and then we show how they can be related to the *Common*Kads model set. In section 4.1.2 we show ways of validating and verifying the models.

4.1 Requirements identification in Software Engineering

Software engineering [Sommerville, 1989] divides the process of requirements identification into three sub-processes;

1. Feasibility study. An assessment whether the project makes sense. Topics investigated are whether the user wishes are satisfiable, whether a system could be cost

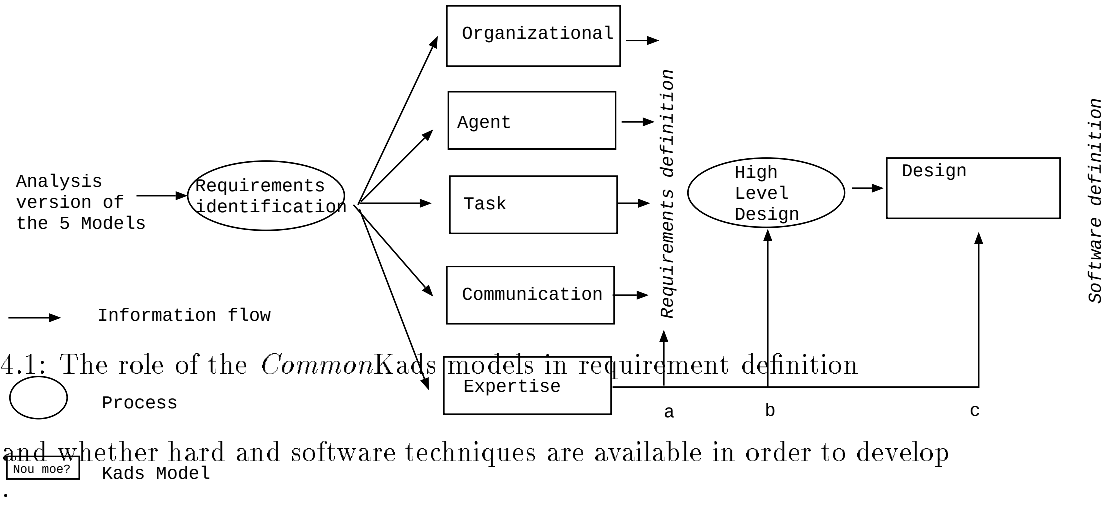


Figure 4.1: The role of the *Common*Kads models in requirement definition

effective and whether hard and software techniques are available in order to develop a system.

2. Requirements definition. This is the process of writing up a definition of requirements in natural language that is readable both by future users and management. It serves as a bidding for a contract for a requirements specification. It also provides an initial idea what tasks the system will perform. In order to achieve this one can perform some *system modeling* which is the creation of a model of the 'real world entities' which are to be represented in the system. Viewpoint analysis, finite state machines and relation models can be part of this analysis in software engineering.
3. Requirements specification (functional specification). The process of developing a detailed and precise description of the system requirements, also called functional specification. The requirements specification is written in a more formal way then the requirements definition and should be detailed enough to serve as a contract between the procurer and the developer. It can either be formal or informal. Formal requirement specifications have the disadvantage that they can often not be understood by the procurer.

The step after requirements specification is the specification of the artifact to be constructed; Software specification. The process of developing an abstract description of the software design (technical design). The technical design describes the realization of the requirements specification in terms of the artifact under construction. Figure 4.2 shows the processes and deliverables that are produced in 'standard' software engineering (after [Sommerville, 1989]).

4.1.1 Requirements definition in *Common*Kads

Requirements definition and specification documents define what the artifact should be able to perform. They are created partly by the future users and for the future users of the system.

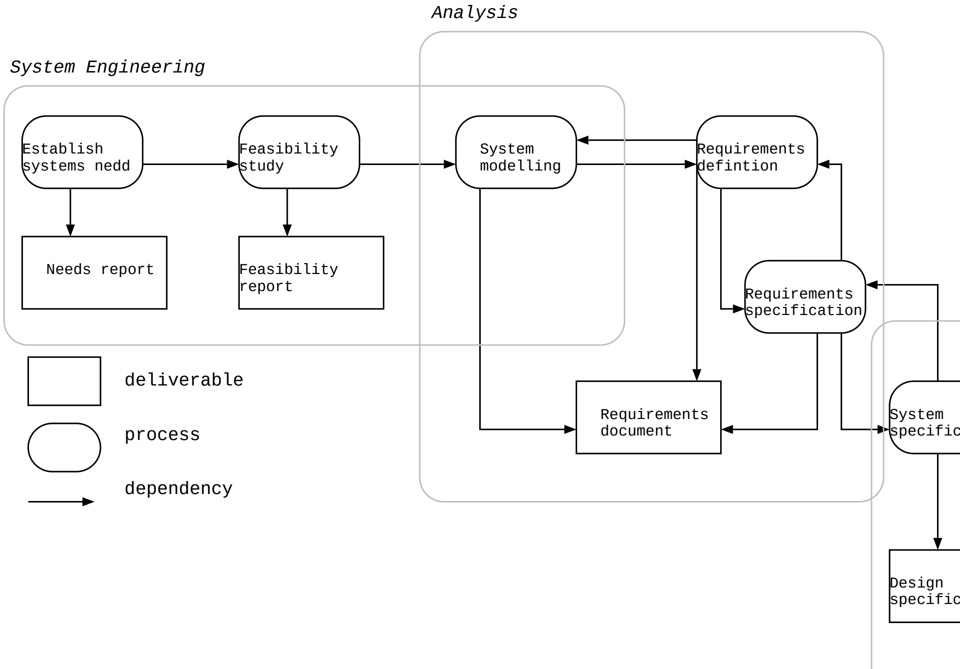


Figure 4.2: Processes and deliverables part of specification in software engineering

In *Common*Kads requirements are distributed over five models; task model, agent model, expertise model, communication model and the organizational model.

Figure 4.3 shows a diagram of the dependencies between processes in the methodology and models. The general direction of the whole process is top down and follows the dependencies, but the way in which information is achieved is not necessarily in a top-down fashion. The is due to the dependencies that exist between all the pieces of information in this figure and also the available expertise and technology. The approach to the development of the models is 'risk' driven [Taylor, 1992]; this means that it is advised by the methodology to work on those tasks that have the highest risk of giving development problems first. The figure shows that the methodology support consists of generally applicable parts (generic models), guidelines, and a set of models in which results can be stored in a systematic way.

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 [Vanwelkenhuysen, 1991] an expert system has been created that diagnoses printed circuit boards. 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.

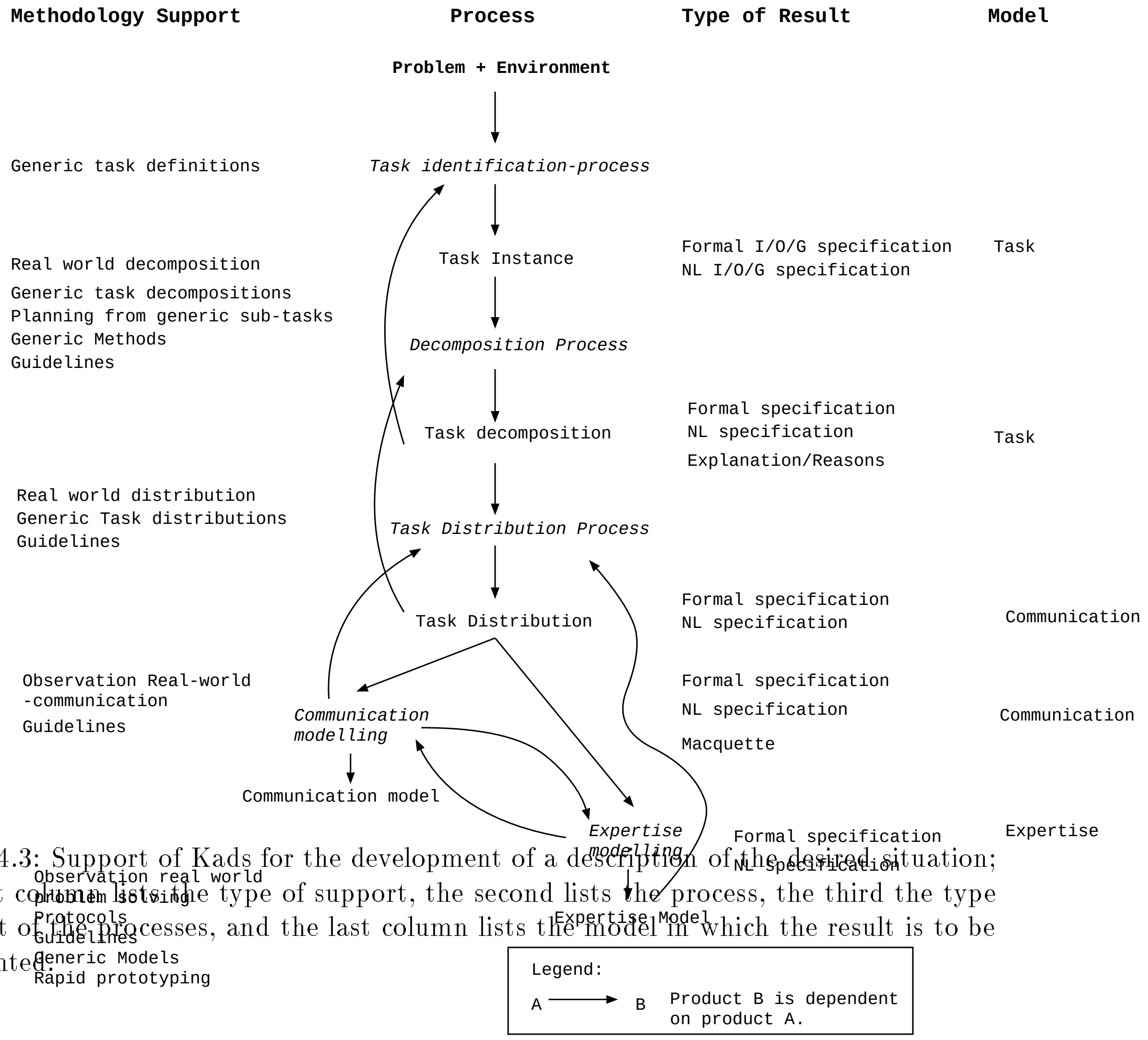


Figure 4.3: Support of Kads for the development of a description of the desired situation; the first column lists the type of support, the second lists the process, the third the type of result of the processes, and the last column lists the model in which the result is to be represented.

4.1.1.1 Task model

The task model [Duursma, 1992, Martil et al., 1992] "describes the tasks that are executed in the organizational 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 at least input, output and goal.

Various instantiations of the task model can be made during a project, e.g. describing existing situations, past situations, desired situations. Only the task models that described the desired situation are relevant as specifications and form an input input for the design process. The task model is a means to organize these models of work.

The task model of the desired situation provides a structured a set of goals to be achieved by the system and the user. An instantiated task model of the desired situation can be part of a of an initial system model, a rough description of the functionality of the system to be delivered. During requirements study it can be elaborated to be part of the (functional) requirements definition. In this case it needs to go together with a communication model.

4.1.1.2 Agent model

The agent model [Martil et al., 1992] 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.

4.1.1.3 Communication model

The communication model [Martil et al., 1992] (see also section 4.2) 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 and Breuker, 1989a] 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.

Because of the fact that the communication model has not the state of development as the expertise model we elaborate it in this document. Our elaboration is based on the model of cooperation in KADS-I.

4.1.1.4 Expertise model

The expertise model can be applied during the analysis stage and during the design stage. During analysis it is used to get an overview of the reasoning and the domain knowledge applied on the task by human experts. In design it is used to make a high level design of the inferencing and domain knowledge of the system. In software engineering data flow diagrams and data dictionaries are applied for this purpose, but these are not as expressive as the four layer model.

The expertise model [Martil et al., 1992, Wielinga et al., 1992a] refines the functional specification of the tasks assigned to the system by identifying a high level 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'.

It is therefore a topic of discussion whether the expertise model should be part of the requirement definition (arrow 'a' in figure 4.1) or an input to high level design (arrow 'b') or whether it is a high level design already (arrow 'c').

Arguments in favor of the use of the expertise model in a requirement definition:

- elaborates the functional specification,
- Systems of this complexity cannot be specified easily in a behavioural way.
- Expertise model plus principle of rationality prescribe a class of behaviours.

Arguments contra the use of the expertise model in a requirement definition:

- The elaboration of the functional specifications deviates from 'normal' functional specifications because it describes a high level 'how' instead of a 'what'.
- Unique interpretation of behaviour is impossible for a client because the principle of rationality is ambiguous and can therefore not serve as a (legal) contract.

Though it has been argued that there is no fundamental difference between a complete behavioural description and a full model with interpretation function that can create the description [Hoffmann, 1992] one model can certainly be more convenient than another one. The personal opinion of the author of this chapter is that the expertise model should not be part of a requirements definition because of the reasons mentioned above. However, we have included the expertise model in the model validation & verification sections of this chapter in order to be complete.

4.1.1.5 Organization model

The organization model represents organizational issues. From the organization model the most non functional requirements definitions can be derived. This issue has not been elaborated in this document.

4.1.2 Requirement validation & verification

Requirements should be formulated in such way that the design can be validated against the requirements. Earlier in the project a set of definitions proposals were collected [Tomlinson, 1991]. Here we quote some of the relevant ones again:

IEEE provides the following definition:

> Validation: The process of evaluating software at the end of the development process to ensure compliance with software requirements. (*IEEE No. 729-1983*)
>
> Verification: The demonstration of consistency, completeness and correctness of the software at each stage and between each stage of the software development cycle, [Branstad and Cherniavsky, 1982]

In figure 4.4(from [Ayel and (editors), 1991]) an adaptation of these concepts is shown for knowledge based systems. In this figure validation is performed by the customer to test whether the system is usable; so both the requirements and the actual system are being tested; while verification is done by the builder to test whether the software is correctly implementing the specifications.

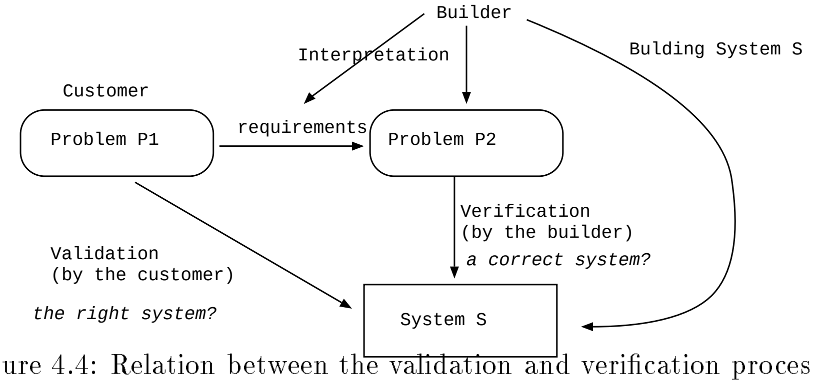


Figure 4.4: Relation between the validation and verification process

Also in KEATS, the knowledge engineer's assistant, these definitions have been adapted for Knowledge Based Systems;

> In Knowledge engineering validity should provide a measure of how faithful the system's architecture and behaviour are with respect to the problem area [Motta et al., 1989]
>
> Verification is the process of assessing the consistency, correctness, and completeness of a knowledge base ...While correctness and consistency checking normally depend on the formalism being used, completeness is very much a semantic property depending upon the task or the problem solving method being employed. [Motta et al., 1989]

It is essential to reduce the risk of designing by applying continuous validation and verification. In this section we specialize the validation and verification issues above in order to be used for the *Common*Kads models. In line with the quotes above, we will use the concept of verification for syntactical consistency checks, and validation for checks concerning the validity of the semantics of the model in relation to the problem area. Verification includes testing for (internal) consistency, (internal) completeness. Validation refers to (external) usability and acceptable behaviour. Validation can either be done by building, testing and rebuilding a system, like in spiral development models, by applying validation guidelines, or by empirical validation methods that allow the assessment of the validity of the requirements at a certain stage of development. Verification can be handled by formal means, like a calculus, or logics, or in an informal way by guidelines.

Verification by the use of a formal calculus has been elaborated for the expertise model only. Consistency of the expertise model can be assessed up to a certain extend in this way. The formalization of the expertise model can also be used for some empirical validation because parts of it can be operationalised. Care must however be taken that the formal model is a valid copy of the version in natural language. The structured natural language (CML) version is used for communication between other models, thus in the end we are interested in the validity of this version.

Other models can be checked by some guidelines and some can be prototyped, like the communication model and parts of the design model.

In *Common*Kads we can distinguish the intra and inter verification of the input models; is the model internally consistent and complete, and is the relations with the other models consistent and complete Each of the models should provide a method to perform the internal verification. This method can either be a set of guidelines or a formal verification.

The validation process in *Common*Kads can be carried out for each of the models separately. All models have an impact on the problem area, so all need some for of validation. In this chapter we have analyzed validity and verification issues concerning task, communication and expertise model in some depth, and the dependencies between all the models.

4.1.2.1 Intra model verification & validation

Validation is a difficult issue in some of the *Common*Kads models. The easiest are the formal version models. Consistency of the formal models can be be tested with automatic support. Only interpretation of the meaning of the formal models in terms of the real world can still be problematic. The non-formal models are difficult to verify. They are not formal and neither in pure natural language, but in a ontology of concepts of which the meaning should be clear. In conventional software engineering the requirements are either specified in (structured) natural language or in specification languages that have a formal semantics.

A completely different problem is caused by the nature of expert systems; they have usually impact on the organization. It is often very difficult to fit an expert system into the organization. The consequences of the introduction of the system are hard to foresee. Therefore the requirements are difficult to formulate. This makes the use of prototyping or maquettes for requirements necessary as a method of validation.

In the next few paragraphs we go through the models and investigate their consistency and completeness. For each of the models a set of validation guidelines is under development.

---

[0] The issue of validation will be addressed in greater detail in the Quality Viewpoint document and the Maintainability document

Task model -*consistency* The consistency of the task model can be checked manually by checking the match of the input/output of subtasks on the input/output of parent tasks. Secondly it can be checked whether the achievement of the goals of the sub-tasks will lead always to the achievement of the parent task. If a formalized task model was available of the task model, and then consistency could be checked by unifying the goals of the parent tasks with the combination of the goals of the sub-tasks.

Task model - *Completeness* The completeness of the task model has two aspects; (1) the tasks addresses the correct tasks to be solved in the organization, (2) the recursive decomposition of sub-tasks is complete. The first issue be checked by reading the input, output and goal specification of the root task. Input and output of the root task is elaborated in the decomposition. Therefore the complete tree has to be checked for a valid mapping on the real problems to be solved by the system.

Example *Task model validation* In the flight planner case the top level task can be described as 'flight scheduling'; the top level task is defined as:

task *Scheduling*
agent
Scheduling department
input
Flight program
output
Crew assignments
goal
To cover all flights of the flight program with minimal
costs.

This task definition cannot be decomposed by itself; more analysis and a more refined definition of the tasks to be carried out by the system is taking place in investigating the subtasks; for instance

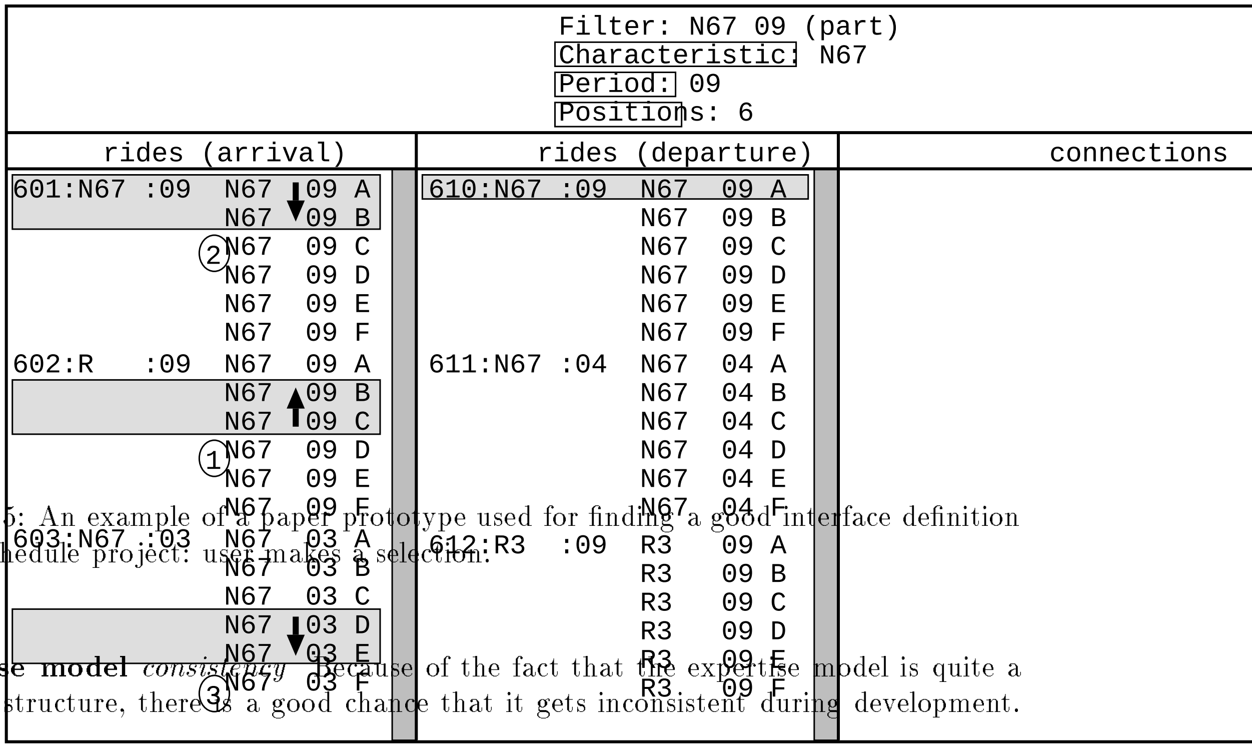
task *Planning*
agent
The planning department; which are two persons.
input
The flight program.
output
The amount and type of personnel required and a schedule.
goal
The goal is to create a schedule with minimal cost that
respects all aviation regulations. The amount and type of
personnel should be able to fly this schedule.
decomposition
Plan medium haul, plan long haul.
task-decomposition-type
The type of task decomposition here is the separation of
independent goals (modelling guideline)

Communication model *consistency* The internal consistency of the communication model can be assessed by checking dependent attributes.

Communication model *completeness* The completeness of the communication model can only be assessed by relating it to the task and expertise model. The only check that can be made locally is by investigating whether the tasks in the communication model have a value assigned to it.

Communication model *usability* Usability is the most important aspect of the communication model. Maquette based approaches are centered around this aspect. In order to be able to test the usability of the interface prescribed by the communication model it has to be realized in some way; either by 'Wizard of Oz' approaches for command language grammar interfaces or implemented in fast interface development environments like Hypercard, Interlisp, or tools like those that can be found on Symbolics Common lisp environments.

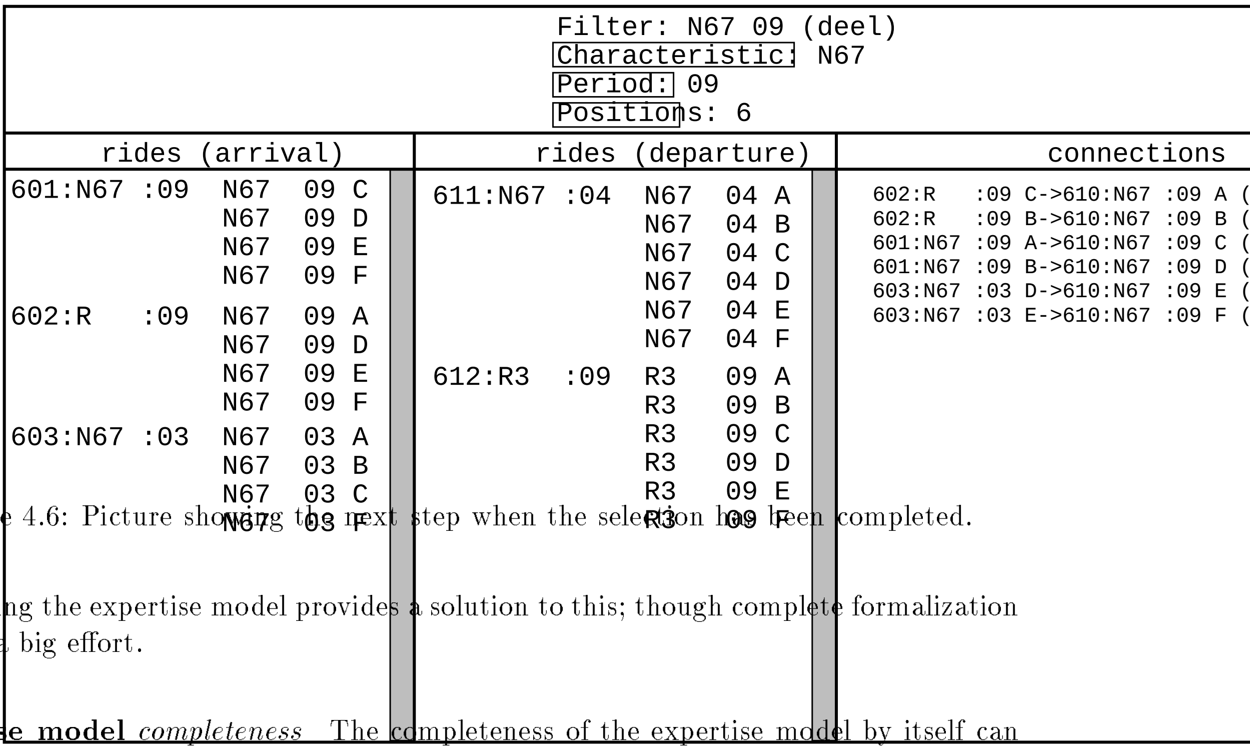
Example Figure 4.5 shows a picture that was used as part of a a set of pictures to discuss the interface design of the Scheduling support system for the Belgian Railways. The arrows in the picture show the direction of a selection process; three parts of an incoming train are selected to form part of an outgoing train. The next picture (figure 4.6) shows that the selection has been completed, and the result has been shown in the connections window. Interesting is that in previous prototypes the interface was more graphical; trains and connections were represented by lines between different stations, but the users preferred a system that required less 'looking at the screen' and less mouse handling. It is difficult to predict these responses to your interface design, and therefore a maquette is a good way to reduce the risk of building an unusable system.



```
                                   Filter: N67 09 (part)
                                   Characteristic: N67
                                   Period: 09
                                   Positions: 6
    rides (arrival)          rides (departure)              connections
601:N67 :09  N67 09 A    610:N67 :09  N67 09 A
             N67 09 B                 N67 09 B
           ② N67 09 C                 N67 09 C
             N67 09 D                 N67 09 D
             N67 09 E                 N67 09 E
             N67 09 F                 N67 09 F
602:R   :09  N67 09 A    611:N67 :04  N67 04 A
             N67 09 B                 N67 04 B
             N67 09 C                 N67 04 C
           ① N67 09 D                 N67 04 D
             N67 09 E                 N67 04 E
             N67 09 F                 N67 04 F
603:N67 :03  N67 03 A    612:R3  :09  R3  09 A
             N67 03 B                 R3  09 B
             N67 03 C                 R3  09 C
             N67 03 D                 R3  09 D
             N67 03 E                 R3  09 E
           ③ N67 03 F                 R3  09 F
```

Figure 4.5: An example of a paper prototype used for finding a good interface definition in the Schedule project: user makes a selection.

Expertise model *consistency* Because of the fact that the expertise model is quite a complex structure, there is a good chance that it gets inconsistent during development.



```
                                        Filter: N67 09 (deel)
                                        Characteristic: N67
                                        Period: 09
                                        Positions: 6
     rides (arrival)          rides (departure)              connections
601:N67 :09  N67  09 C    611:N67 :04  N67  04 A    602:R   :09 C->610:N67 :09 A (
             N67  09 D                 N67  04 B    602:R   :09 B->610:N67 :09 B (
             N67  09 E                 N67  04 C    601:N67 :09 A->610:N67 :09 C (
             N67  09 F                 N67  04 D    601:N67 :09 B->610:N67 :09 D (
602:R   :09  N67  09 A                 N67  04 E    603:N67 :03 D->610:N67 :09 E (
             N67  09 D                 N67  04 F    603:N67 :03 E->610:N67 :09 F (
             N67  09 E    612:R3  :09  R3   09 A
             N67  09 F                 R3   09 B
603:N67 :03  N67  03 A                 R3   09 C
             N67  03 B                 R3   09 D
             N67  03 C                 R3   09 E
             N67  03 F                 R3   09 F
```

Figure 4.6: Picture showing the next step when the selection has been completed.

Formalizing the expertise model provides a solution to this; though complete formalization requires a big effort.

Expertise model *completeness* The completeness of the expertise model by itself can be assessed by checking the relations between the levels. Every inference need domain knowledge that can be lifted to it, every required piece of domain knowledge need to be available in the application domain. A method need to be available for every inference structure.

Expertise model *appropriateness* An input for the development process of the expertise model is for instance a protocol of experts solving the problem. This protocol is than interpreted in terms of elements of the expertise model. However, the problem solving behaviour (the protocol) is based on a different interaction with the world the the expert system will have. Therefore there is no guarantee that the problem solving model will also work satisfactory in a system (arrow c in figure 4.1) [Rademakers and Pfeifer, 1992, Hoffmann, 1992]. Though this problem poses limits to the reliability of the expertise model as a high level design, it is practically useful as a communication medium between domain expert and knowledge engineer.

The expertise model a first step to the technical realization of the tasks to be performed by the system. From reading the expertise model we can get an idea what the system will do. This 'reading' is however informal; the processes still have to be turned into algorithms and have still to be coded. Therefore the expertise model cannot provide any guarantee about the practically feasible realization of the prescribed behaviour; it may appear that the algorithms found for the expertise model require unreasonable time or space.

Example *Expertise model is no guarantee* In the Case study at the end of this deliverable the selection of suitable combinations has been represented by a 'select' inference action. In the design process an algorithm has to be found that realizes this select inference. We reduced the select inference to a standard set covering problem and identified and coded an algorithm for it. When we looked at the behaviour of the program, it appeared that the combinations constructed were in some respect different then those created by the expert. The reason for this was that we had lost information by the reducing the inference to a set covering problem. Apparently more information is used by the expert in order to construct his flight combinations, like delayed flights due to mist or flight congestion.

4.1.2.2 Cross model verification

In *Common*Kads the requirement specifications are distributed over several different models. These models should be consistent before starting the design process. For instance the task model should fit on the organizational model, the expertise model should fit on the communication model. Unfortunately these models cannot be developed completely independently from each other. Though the interfaces between the models are well defined, the requirements of each model depend on the requirements of one or more other models. The next chapter of this report discusses the process in which each of these models can be developed.

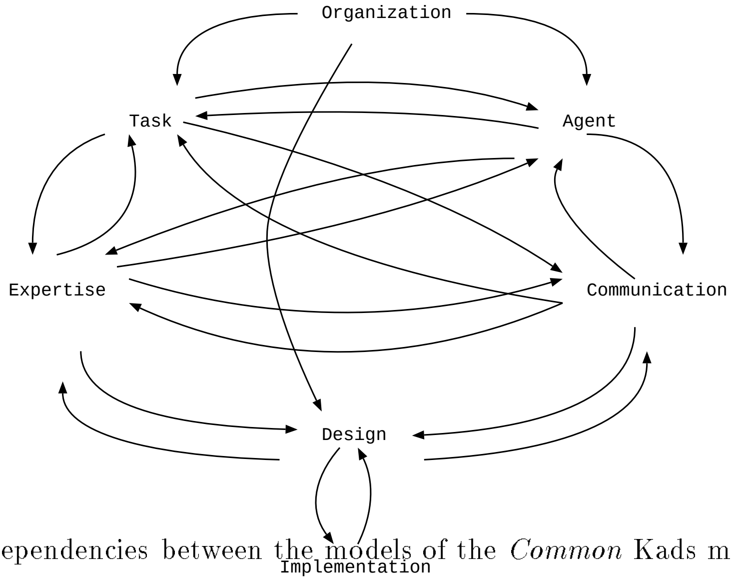


Figure 4.7: The web of dependencies between the models of the *Common* Kads methodology

Task ← Organization

Task ← Agent

- Task hierarchy should reflect the problem to be solved in the organization.

Task ← Communication

- All point where there is a transfer of input output between agents should be included as transfer tasks in the communication model.

Expertise ← Task

- The method competence should comply with the task goal.
- The method assumptions should hold for the task (also related to organization model).

Expertise ← Design

Expertise ← Communication

- All tasks assigned to the system in the Communication model should be elaborated in expertise model.
- The expertise model should provide sufficient detail to provide the architectural basis for different types of explanation.
- The Domain knowledge should be sufficiently well defined to make knowledge stuffing possible.

Expertise ← Organization

- Knowledge required for the methods available at acceptable costs in the organization.
- Knowledge maintenance is feasible in the Organization at acceptable costs.

Expertise ← Design

- Every method, inference and task structure should be realizable by an algorithm or by a set of heuristics.
- The different domain ontologies should be supported.
- All domain knowledge of the domain layer of the expertise model should be included in the design.

Communication ← Expertise

- Data acquisition for all methods should be handled.
- Knowledge acquisition (stuffing) for all knowledge categories in the domain layer should be handled.

Communication ← Organization

- Task distribution should fit in the organization.
- Agent relations should fit in the organization.

Communication ← Agent

- The agents that perform the the tasks in the Communication model should have the capacities for performing those tasks.

Communication ← Design

- The architectures for the types of explanation should be feasible.
- The data entry and presentation should be feasible.
- The knowledge acquisition (stuffing) support tools should be feasible to design.

Design ← Organization

- The chosen architecture and/or language is available or can be made available within the organization.
- The required hardware can be provided by the organization.

Design ← Expertise

- The design should cover all applied methods
- Design should cover all domain knowledge

Design ← Communication

- Architecture should support the type of explanations required.
- Architecture should support the type of data entry and presentation required.
- Architecture should support the type of knowledge acquisition required.

Design ← Implementation

- The Architecture should have an 'easy' mapping on the chosen implementation formalism.
- The design should allow some structure preservation of the design in the code.

4.2 Communication Requirements

Due to the fact that the communication model is not as far developed as the model of expertise we elaborate here the communication model; as it represents the communication requirements to the system to be designed. We take as input the model of cooperation of KADS-I and complete it to the following categories of requirements:

4.2.1 Elements of the communication requirements

- **Task distribution**. The distribution of tasks between the different agents involved.
    1. **Fixed assignment**. A task assignment that is static; every tasks is always performed by the same agents.
        - (a) **One task - One agent.** The task is performed by one agent only.
        - (b) **One task - several agents.** The task can be performed by several agents. This is the case in advisory/supervision agent relations (see below).

2. **Flexible assignment.** The assignment is dependent on the problem solving process. This can either be problem solving (impasse) driven by a system agent, or user driven.

- **Task concurrency**
  - Parallel executed tasks. This represents that tasks can be performed in parallel by different agents. In multi agent systems this is a common case. In multi process or parallel problem solving this can also be the case.
  - Single process sequentially executed tasks

- **Task control (initiative).** This can be one of:
  1. Assigned initiative. The initiatives are assigned at design time.
  2. Execution time determined initiative.
     - (a) Discourse planning. The initiative is decided upon at execution time. In this case the requirements, expertise model and design of the discourse planning need to be elaborated in the task, expertise and design models.
     - (b) Open, data driven initiative. The initiative is dynamically decided at execution time depending on the incoming data and the state of the problem solving process.
     - (c) Open, user initiated initiative. The user performs the initiative and sets tasks to perform some work; but keeps control herself over the overall process.

- **Assumption & Competence validation.** This item refers to the check whether tasks are still the competence of the system. This depends on the method competence and the available knowledge. In the PPS approach this problem is explicitly dealt with (see chapter 3). This can be either be
  1. No assumption validation by system, knowledge acquisition on user initiative.
  2. Continuous assumption validation and integrated knowledge acquisition (PPS).

- **Explanation**

  Explanation has been divided in three categories by Chandrasekaran [Chandrasekaran, 1989], all belonging to the 'introspection' type of explanations.
  1. No explanation.
  2. Explaining decisions (type 1).
  3. Justifying the Knowledge base (type 2).
  4. Explaining strategy (type 3).
  5. Outside the 'introspection' type we find Explanations without reference to problem solving activity, these can be static ('canned text', like emacs texinfo or Unix man pages) or adapted to the user knowledge.

- **Data Manipulation**

  1. Direct manipulation
     (a) Data visualization
     (b) Selection
     (c) Abstraction
     (d) Filtering
  2. Manipulation via Command Grammar.

- **Command formulation**. How are the commands between the agent formulated. Most of the complexity if of course for user-system interaction. In general we can make a distinction between;

  1. Selection (e.g. menu, icon, bar-code-pen, touch screens)
  2. via Command language

- **Data acquisition**

  1. System initiative
     (a) Logical Specification driven dialogue.
     (b) Problem solving driven.
  2. User Initiative
  3. Mixed initiative (Data blackboard).

- **Agent relations.** Agent relations cover a set of communication attributes. This information can also be represented in the agent model. The agent relations provide an entry to set at once a whole set of communication attributes in the communication model; (1) the responsibility of the task, (2) the fact whether a task is executed by one agent only or by two or more agents and results are compared [Kjaer-Hansen, 1991], (3) The initiative for the task;

  1. Instructor. One agent performs the instructor role and another agent the student role. This implies that the task involved is performed by at least two agents.
  2. Informer. The informer agent proposes a first shot for a solution of the task, but has no responsibility for the quality of the proposed solution.
  3. Advisor. The advisor agent advises a certain solution for the task. The advisor proposes a finished solution but has no responsibility for it.
  4. Questioner (?)
  5. Commander. The commander agent requests the execution of another task.
  6. Supervisor. The supervisor agent checks the task results of supervised agent.
  7. Colleague. A set of agents have equal responsibilities for the task.
  8. Autonomous

4.2.2 Finding communication requirements and agent relations

It is very difficult to predict how a system, once realized will change the user environment. Therefore several methodologies have proposed a prototyping for requirements approach. In [de Greef, 1989] this the 'Wizard of Oz' technique is applied; an operationalisation of the communication model where the problem solver is played by a human expert. This works well for teletype kinds of interfaces but not for window and menu based interfaces. For modern interfaces a maquette can be built to test the proposed interface and the requirements to the functionality of the system to be constructed. 'The maquette based methodology' [Alaman et al., 1992] centers around this issue.

At the AI lab of the Free University of Brussels it has been noted in two recent projects that a requirements prototyping is essential for producing the requirements of a system that will work satisfactory in an organization. The process has been called 'situated adaptive design' [Rademakers and Pfeifer, 1992]. The development of the 'Schedule' system for the Belgian railways shows that the development of a good task distribution, initiative and interface design is essential for the acceptance of a system. It appeared that requirements on the expert system component can only be formulated when some stable, accepted, interface system has been formulated.

Problematic is that the requirements of an interface and the problem solving functionality of an expert system are not independent. If the interface is a very handy data modelling tool, the user might have such a good overview that intelligent support is not necessary anymore. Otherwise, if the expert system works perfectly for all cases, the interface is less relevant and the user can just note the results.

One solution for this problem is to have flexible task distribution between problem solver and user. This means that the user-system combination can perform a task in several ways. This leaves more freedom to the final users of the system. Requirements can still be specific for the different approaches that are to be included in the system.

Can a complete behavioural description be created of a problem solver? By definition problem solvers in AI cover a large set of problem-solution pairs that can in practice not be enumerated beforehand. But in the same way the queries of a database cannot be enumerated beforehand. A abstraction has to be made of classes of data that are the input data and the output solutions. A problem is here that the database queries are formulated in a defined language that can be included in the specifications. Expert systems collect their data often through a dialogue with the user. The structure of this dialogue is important for acceptance of the system. In MYCIN there are famous examples where the order of the questions was exceptionally strange, and where un-appropriate questions from a social point of view were asked in the process of performing a diagnosis. These problems should be investigated within the requirements definition are in *Common*Kads documented in the Communication model.

Chapter 5

Principles Guiding the Design of the Problem Solver

Guus Schreiber

5.1 Structure-Preserving Design

As pointed ou in Ch. 3, design consists of (i) the specification of an system architecture and (ii) the selection of appropriate representations and computational techniques (the defintion of the virual machnie(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 artefact. 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. 5.1.2 we discuss the main advantages of the structure-preserving approach.

In Sec. 5.2 we take the "executable specification" view on the expertise model and try to define skeletal architectures for supporting structure-preserving design of the problem solver. In Sec. 5.3 we investigate how the structure-preserving approach influences the computational decisions.

5.1.1 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 outlined in Ch. 3 (data, function, control).

Functional perspective

The functional 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 domain knowledge represents a specification of all domain-specific data and forms the major part of the data perspective.

The dynamic and staic 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" view) 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 5.1 summarises the information contained in the various parts of the expertise model with respect to three viewpoints.

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.

5.1.2 Advantages of structure-preserving design

There are a number of reasons for following a structure-preserving approach to design:

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 5.1: Characterisation of the information contained in a expertise model with respect to the three perspectives on systems: data, function and control.

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 artefact 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 artefact. 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 → fever = present
diastolic-pressure $\geq$ 95 → blood-pressure = elevated
blood-pressure = elevated → 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. 5.1 depicts a simple inference structure containing these two inference steps.[1] The italic annotations of the dynamic knowledge rolees 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").

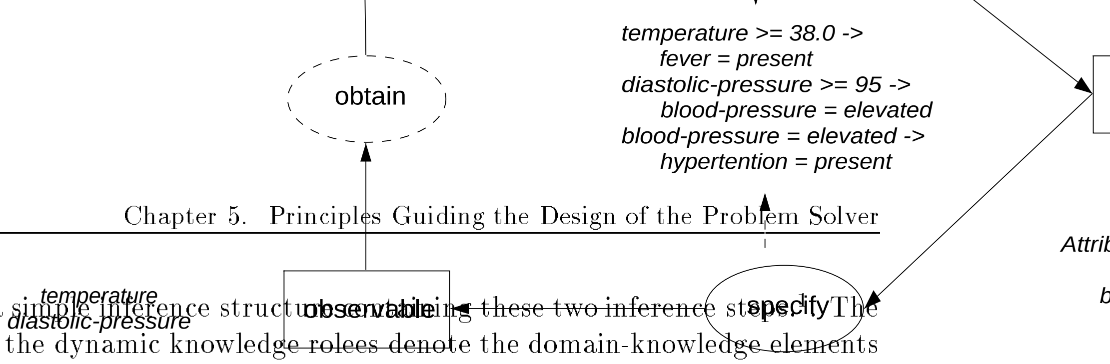


Figure 5.1: Two inferences in a medical application with some examples of related domain knowledge. Both *abstract* and *specify* use the same domain knowledge, but for different purposes. The dashed oval denotes a transfer task.

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 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.

5.2 Structure-Preserving Design: A Skeletal Architecture

In structure-preserving design, the idea is to use the expertise model as a skeletal architecture of the artefact. In fact, most knowledge engineers using COMMONKADS think

---

[1] This inference structure is in fact a small fragment of the KADS inference structure for heuristic classification (see [Schreiber, 1992b]).

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 input/output functions:

Inference method

Inferences specify the nature of the input and output (the dynamic knowledge rolees) and the domain knowledge used in deriving the output from the input (the sratic 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 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, 1992c] 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 rolees) 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

---

[2] If the expertise model is fully specified in some strictly formal language such as ML^2 (cf. [van Harmelen and 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 . In task 1.2, a theorem-prover is being 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.

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 rolees 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.

HCI 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. 5.2 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 control knowledge as specified in the model. Typically, it influences the design of task knowledge and strategic knowledge specified.

Fig. 5.2 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.

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

---

[3] The skeletal architecture makes the simplifying assumption that the external agents the system will deal with are all humans.

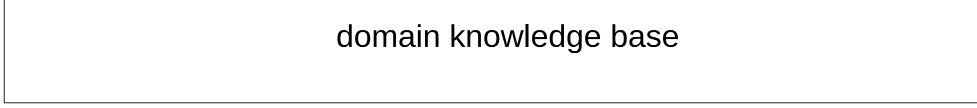


Figure 5.2: Skeletal architecture supporting structure-preserving design. The boxes with rounded corners denote system components that are active during reasoning; the sharp-edged boxes denote data stores. Non-dashed lines describe a control operation (*activates*) between components; dashed lines a data operation (*store*, *retrieve*, *interpret*).

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. 5.3, 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, 1992c].

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. 5.3.

5.2.1 Meta-level vs. object-level inferencing

The skeletal architecture shown in Fig. 5.2 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. 5.2 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.

5.3 Structure-Preserving Design: 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].

5.3.1 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 and 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. 5.1. A full listing of the code, together with some sample traces, can be found in Appendix A.

Example fragment of a expertise model We assume that the inferences in Fig. 5.1 are specified in the following way in the conceptual model:

inference *abstract*
  input-role:
    finding → some expression about patient-data
  output-role:
    finding → some expression about qualitative-data
  static-role:
    abstraction-knowledge →
      < relation(qualitative-abstraction), relation(definition) >

inference *specify*
  input-role:
    finding → some expression about qualitative-data
  output-role:
    observable → some property of qualitative-data

---

[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.

static-role:
specification-knowledge →
< relation(qualitative-abstraction), relation(definition) >

These inferences use the same domain knowledge (see the staic role), but for different purposes.

The domain knowledge is specified in the expertise model using the domain description language (DDL) defined in [Schreiber et al., 1992]:

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) ;
tuples:
< temperature ≥ 38.0, fever = present >
< diastolic-pressure ≥ 95, blood-pressure = elevated > ;

relation *definition*
argument-1: expression(qualitative-data) ;
argument-2: expression(qualitative-data) ;
tuples:
< blood-pressure = elevated, hypertension = present > ;

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 DDL statements onto the chosen knowledge representation. This representation was also used for the implementation of the Sisyphus application [Schreiber, 1992c]. 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 DDL 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)
% dyn_role(Inference,     Input/Output,        General name, Specialised name).
% static_role(Inference, , Inference knowledge).

inference(  abstract,  'Abstract').
dyn_role(  abstract,  input(1),        finding,     'Specific finding').
dyn_role(  abstract,  output,          finding,     'General finding').
static_role(abstract,  relation(abstraction, finding, finding)).
```

```
inference( specify,   'Specify').
dyn_role(  specify,   input(1),  finding,     'Finding to be clarified').
dyn_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. 5.3 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 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 A.

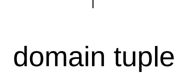


Figure 5.3: Schematic overview of a sample execution of the inference *specify*.

5.3.2 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. 5.5.

   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 StatCons 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 rolees 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 A an example control structure representation can be found for two tasks that apply respectively the *abstract* and the *specify* inference.

5.3.3 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. 5.2, 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. 5.2 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. 5.2.1).

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].

5.4 Existing Approaches to Computerised Support

Although in principle a knowledge-level model can be viewed as a specification that can be implemented in a conventional manner, for most modelling approaches dedicated environments exist, that support structure-preserving operationalisation. In this section, we discuss the nature as well as the merits and limitations of each of approaches.

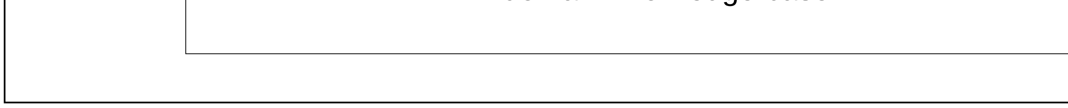


Figure 5.4: Skeletal meta-level architecture in which the strategic knowledge is operationalised through an separate meta system. This meta system uses a a causally connected knowledge-level description of the object system to reason about and act upon this system.

5.4.1 Types of support environments

Existing support environments can roughly be be divided into three categories:

1. Task-specific shells
2. Task-specific programming languages
3. Task-independent programming languages

Task-specific shells Task-specific shells support the operationalisation of a range of application domains. In COMMONKADS terms, a task-specific shell can be seen as an operationalisation of an interpretation model. The model incorporated in a task-specific shell represents a problem solving method for solving a certain type of problems. Example task-specific shells are MOLE (method: cover & differentiate; [Eshelman, 1988b], SALT (method: propose & revise; [Marcus and McDermott, 1989]) and **OPAL** (method: skeletal plan refinement; [Musen et al., 1987]). In task-specific shells the structure of the task is fixed: the knowledge engineer cannot change the structure of the domain knowledge (i.e. the domain schema), the set of inferences or the control imposed on these inferences. Only the domain-specific knowledge needs to be entered in the predefined format.

It is assumed that the expert is capable of entering this domain knowledge directly into the system with the help of a dedicated domain-knowledge editor. [Musen et al., 1987] makes the point that this can best be achieved by providing the expert with a support tool that communicates with the user in a domain-specific vocabulary. **OPAL** uses this approach. This is not the case in **MOLE** and **SALT**, where the interaction is (at least partially) in terms of functional objects.

The **PROTEGE** system [Musen, 1989] overcomes some of the problems associated with task-specific shells. This system allows the knowledge engineer to tailor a skeletal planning method to meet the demands of a particular application domain (within the area of managing protocols for medical treatment) and then generates an **OPAL**-like shell that can interact with the expert to acquire the domain-specific knowledge.

Task-specific programming languages A second type of support environments is represented by the task-specific programming languages such as developed within the Generic Task (**GT**) approach [Chandrasekaran, 1988a]. These programming languages contain constructs specific for operationalising a particular (generic) task. An example of such a language is **CSRL** [Bylander and Mittal, 1986], which supports the operationalisation of hierarchical classification problems.

The methodological viewpoint behind the **GT** approach is that problem solving consists of a relatively small set of information processing tasks (the generic tasks) and that a particular instantiation and configuration of these tasks can be used for realising a particular application. Each programming language supports such a generic task. The resulting generic task programs are integrated into one system in an object-oriented fashion.

Task-independent programming languages Task-independent programming languages are high–level programming languages that allow a (relatively) simple mapping from the knowledge-level model onto computational constructs in the language, For **KADS**(-I) a number of such languages have been developed: e.g. Model-K, [Karbach et al., 1991] **OMOS** [Linster, 1992] and **KARL** [Angele et al., 1991]. For the Components of Expertise approach the language described in [Vanwelkenhuysen and Rademakers, 1990b] has been developed. **ZDEST-2** [Tong et al., 1988, Karbach et al., 1988] can also be viewed as such a language, although it is not tied to a particular knowledge-level modelling approach.

The mapping from model to language elements is in these languages usually supported by giving the computational constructs names similar to the model elements. For example, Model-K offers computational constructs with **KADS**-specific names such as *knowledge source* and *meta-class*.

5.4.2 Merits and limitations of the approaches

Task-specific shells provide a high level of support to the knowledge engineer. If the application domain fits well with the environment, then only relatively little effort is necessary for building a system. The price paid for the high level of support is a low level of flexibility: small mismatches between domain and tool can already render the tool unsuitable for the target application.

The task-independent programming languages leave the knowledge engineer with considerable freedom during operationalisation. Within the limitations of the computational techniques supported by the language, it is possible to operationalise a variety of applications. On the other hand, their level of support for the knowledge engineer is limited. Unlike the task-specific shells, which contain reusable computational constructs for one particular knowledge-level model, the current programming languages do not contain any reusable pieces of code, e.g. predefined domain structures or inferences. So, the higher level of flexibility is paired with a lower level of support.

The task-specific programming languages take more or less an intermediary position. This approach is flexible in the sense that the control knowledge is not fixed: the knowledge engineer has to 'program" each generic task as well as specify how these should be integrated to solve the overall problem. In terms of support, in particular the reusability of code, this approach is limited: the languages themselves are of course reusable, but there are no predefined pieces of code, such as provided by the task-specific shells.

The three approaches can be seen as as points on a spectrum determined by the level of support vs. the level of flexibility (Fig. 5.5).

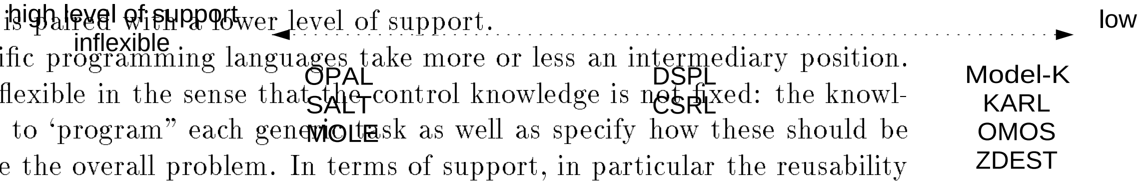


Figure 5.5: Characterisation of some support environments with respect to level of support and flexibility.

5.4.3 Some remarks about operational "knowledge-level" languages

In the last few years, several languages have been proposed for operational and/or formal representation of KADS-type expertise models. Some examples were given earlier in this section. Some additional remarks about this line of research are in place.

It is clear that it is desirable to have, as early as possible in the development process, some machine-executable version of the expertise model. Such prototypes can be used for simulation of the problem-solving behaviour that is specified in the model and can thus play a role in validation of the model.

One should however separate two fundamentally different aspects of such languages:

1. The *modelling* aspects of the language: what constructs does the language offer for modelling the reasoning process in the application domain.
2. The *operational* aspects of the language: what is the operational interpretation of the language constructs.

There is a kind of trade-off between these two aspects. During modelling, one wants to be as flexible as possible in specifying the required problem-solving behaviour. One is primarily interested in the declarative semantics of the language. During operationalisation, one needs to restrict the language to expressions that have operational semantics.

The formal specification languages for **KADS**-type expertise models such as ML^2 emphasise the modelling aspects, although they have a (partial) operational interpretation. The task-independent programming languages are much more directed towards operational aspects. The languages **KARL** [Angele et al., 1991] and **FORKADS** [Wetter, 1990] take an intermediate position, emphasising both modelling and operational aspects.

Fig. 5.6 characterises the different languages on a spectrum from formal specification (emphasising modelling) to executability (emphasising operationality).

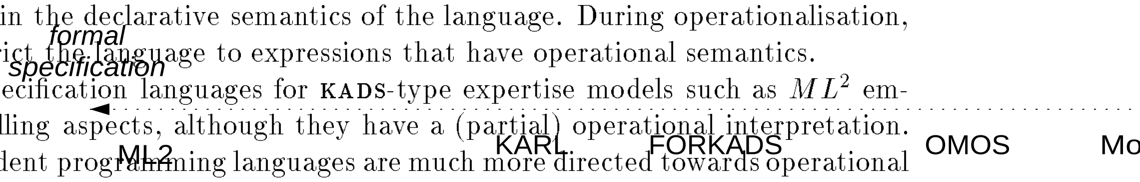


Figure 5.6: Formal languages for KADS expertise models interpreted as points on a continuum from formal specification to executability.

Also, it should be noted that in most real-life applications external requirements dictate constraints on the software environment in such a way that the operational languages cannot be used for the final implementation. For example, the successful Fraudwatch system was implemented partly in Cobol (while following the structure-preserving principle which indeed greatly facilitated the maintenance of the system [Killin et al., 1991]).

5.4.4 Design languages

Another type of support for design can be given by a *design language* such as **DESIRE** [van Langevelde et al., 1992]. In this approach, it is assumed that there exists some (informal) conceptual model of what the system should do. Given this input, the design language supports the formal specification of appropriate system components and their configuration into a system architecture. From this design specification the system code can then be generated automatically.

The **DESIRE** approach fits in fact quite well with the **KADS** approach. One major advantage is that there is a clear separation of roles: the model-of-expertise language used during analysis emphasises modelling aspects; the design language emphasises operational aspects. This circumvents the problems encountered with some afore-mentioned "knowledge-level programming languages" in which the distinction between these, fundamentally different, view points is not clear.

An interesting research question would be to study the possibility of predefining within a design language such as **DESIRE** the components of a skeletal architecture like the one in Fig. 5.2. This would simplify the mapping from the analysis input onto components in the design language.

5.5 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.

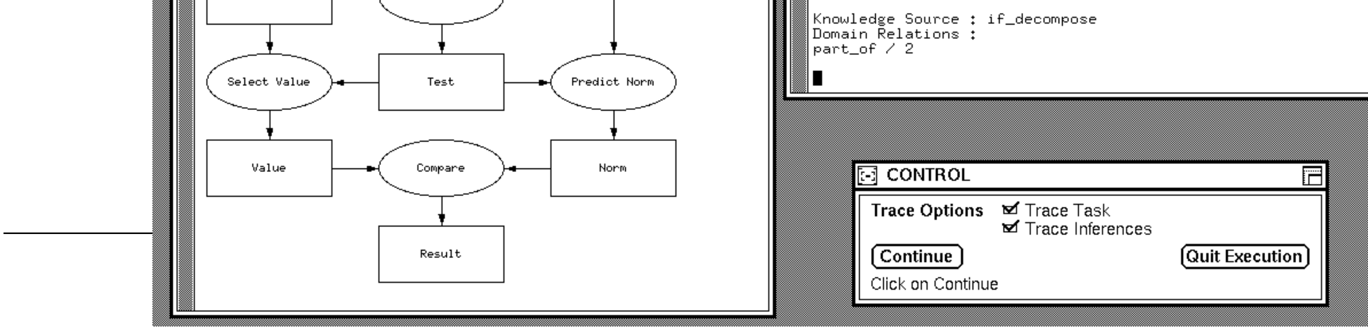
This basic version of the model of systematic diagnosis can be extended in a number of ways. These extensions are adaptations of the model that could be necessary in a particular application. They modify the basic model of systematic diagnosis and define in fact a space of potential systematic diagnosis 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 5.2.

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.
- 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. 5.7.
- 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



```
Knowledge Source : if_decompose
Domain Relations :
part_of / 2
```

Trace Options ☑ Trace Task
☑ Trace Inferences

Figure 5.7: 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 5.2: Inferences in systematic diagnosis models

model. Fig. 5.7 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. 5.2.

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, 1990b], 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]).

Within the context of the Task 1.4 we will develop a more comprehensive environment for supporting the operationalisation of expertise models of scheduling tasks.

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 ideosyncratic for one particular approach and can thus be shared by several groups.

5.6 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, 1992d] provides a rigourous example of structure-preserving design of an office-planning problem solver.

The developers of the Fraudwatch application [Killin et al., 1991] which has been 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. 5.5.

Chapter 6

Principles Guiding the Design of Cooperation

Cuno Duursma

The next step in the development of the system is the specification of the cooperation with the system to be built. Instead of starting immediately with a software specification of the interface, we propose to develop first a description of the cooperation at a more abstract level. This cooperation level is then further elaborated to develop the actual graphical or command based interface.

Cooperation design can be considered to consist of two relatively independent processes; design of the contents of the information to be exchanged and design of the actual interfacing to this information. In KADS [de Greef and Breuker, 1989b] information exchange aspects of the cooperation process have been analyzed in terms of concepts like owners, initiative and ingredients. Here we add the issues of knowledge stuffing, dialogue and explanation.

Figure 6.1: Cooperation design process overview

In this chapter we will discuss the cooperation design, as shown in figure 6.1. We will not discuss the Graphical User Interface (GUI), design or specification (figure 6.2) because we consider this not KBS specific and thus outside the scope of the methodology. process.

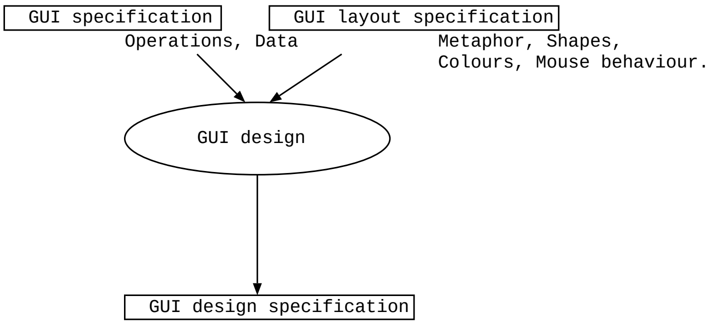


Figure 6.2: GUI design process overview

We discuss here the following items:
Design of:

- Explanation facilities
- Data acquisition
- Knowledge Acquisition

Here we discuss the design process for standard cooperation; knowledge acquisition has not been integrated with knowledge application and the task distribution is fixed.

6.1 Design of Data Acquisition and Presentation

Here we discuss how the system acquires the data for each different case. In the Communication model we have listed the following forms of data acquisition:

- System initiative
    1. Logical specification driven dialogue
    2. Problem solving driven
- User initiative
- Mixed initiative

6.1.1 System initiative data acquisition

The system driven dialogue is the oldest type of dialogue in expert systems. MYCIN, DENDRAL and most other rule-based forward or backward chaining expert system architectures acquire their data at the moment a rule needs that data. In MYCIN this gave rise to inappropriate dialogues form a social point of view, like asking whether a patient is an alcoholic without having asked the age; so children or even baby patients could be asked whether the suffered a drinking problem.

Although not often a good solution, design of system initiative data acquisition is quite simple; the dialogues can be attached directly to the problem solving components, like for instance rules. If the resulting dialogues are not satisfactory, then another dialogue scheme must be specified in the communication model, e.g. a logical specification driven dialogue.

Example *System initiative data acquisition* For a simple rule based (TMYCIN: Tiny EMYCIN ([Jr.,]) with a knowledge base on snakes the interaction looks as follows:

```
> (doconsult)

What is the COLOR of SNAKE86?
    List the colors of the snake.  If there are
    multiple colors, use the following format:
    ((color 1.0) ...), for example,
    ((RED 1.0) (YELLOW 1.0) (BLA{\em Common}Kads 1.0))
    Expected values are: (TAN BROWN BLA{\em Common}Kads GREY GREEN PINK RED YELLOW ORANGE)

((yellow 1.0))
What is the SIZE of SNAKE86?
    Give the approximate size of the snake.
    tiny = 11 inches or less, small = 12-18 inches,
    medium = 19-30 inches, large = over 30 inches.
    Expected values are: (TINY SMALL MEDIUM LARGE)

tiny
What is the PATTERN of SNAKE86?
    What pattern(s) are seen on the snake?
    stripes  = one or more stripes running lengthwise
    bands    = multiple bands around the body
    blotches = large contrasting blotches on the back
    spots    = small, roughly circular spots
    speckles = non-circular contrasting spots
    solid    = solid color without pattern
    If multiple patterns are observed, enter all
    using the format ((feature 1.0) ...)
    Expected values are: (BANDS STRIPES BLOTCHES DIAMONDS SPOTS SPE{\em Common}KadsLES SOLID)

solid
Is RATTLES true of SNAKE86?
    Does the snake have rattles on its tail?
    Expected values are: Yes/No

yes
Is TRIANGULAR-HEAD true of SNAKE86?
    Does the snake have a triangular head,
    noticeably larger than its neck?
    Expected values are: Yes/No

yes
Is BITTEN true of SNAKE86?
    Has someone been bitten by this snake?
    Expected values are: Yes/No

no

The conclusions for SNAKE86 are as follows:

IDENTITY :  RATTLESNAKE (0.80)
```

```
LATIN-NAME :  Crotalus (0.80)

POISONOUS :  YES (0.80)
SNAKE86
```

6.1.1.1 Logical specification driven data acquisition

A specification of a dialogue is like a description of an ordering over the data that can be acquired from an agent (i.e the user). This ordering has to be defined and is in *Common*Kads part of the cooperation model.

When a logical dialogue specification needs to be followed by the problem solver, the problem solving (inferencing) is not completely independent of the dialogue; some data becomes available sooner than actually needed, and some data cannot be accessed before other parts of the dialogue have been processed.

If task control of the problem solver is data driven, then the problem solver can follow the required dialogue as long as new inferences are available that lead to the next step in the dialogue. If the problem solver is based on a fixed control scheme, than this control scheme has to be adapted to the dialogue.

Example *Logical specification driven dialogue* In the snake case above we could define a data acquisition dialogue that prescribes that first should be asked whether a person has been bitten by the snake, at how much time ago before consulting the system, in order to have a really quick diagnosis if there is a chance that the snake is poisonous.

6.1.1.2 User initiative data acquisition

In a user initiative data acquisition dialogue the process is completely under control of the user. The user can provide the data when she wants to. This should go together with a task control that allows this type of interaction, e.g. data driven Task control.

6.1.1.3 Mixed initiative data acquisition

In a mixed initiative dialogue the data acquisition is completely open; a user can put data on a data-blackboard (Musen's whiteboard) and tasks in control take available data or ask for extra information.

Example *Mixed initiative* In the snake example the user could write on the data board e.g. "(COLOR ((BROWN 1.0) (RED 1.0)))" and the system could ask for the value of the PATTERN in order to establish hypothesis like "water snake".

6.2 Design of Explanation

6.2.1 Types of explanation

Basically explanations can be decomposed into three different components [Chandrasekaran et al., 1989a]; (1) The representation of the problem solving activity to drive the explanation process, (2) A filter or translation process to make the translation appropriate for the specific user and (3) an efficient presentation.

In this chapter we will discuss some design approaches to the first component; the introspection of the problem solver in its solution trace to serve as a basis for the explanation contents.

Like we have mentioned in the communication requirements, three types of explanation can be identified (1) trace of decisions, (2) trace of strategy and (3) a justification.

Usually explanation facilities (type 1 to 3) are built on top of the problem solvers architecture. The justification of knowledge requires however access to deep(er) models, the explanations need to be embedded in more general principles in the domain. Some kinds of explanations require explicit representations of problem solving strategies.

6.2.2 Architectures for explanation

In his famous system XPLAIN [Swartout, 1983] Swartout explored a combination of a description of abstract goals, an automatic programmer and an explanation facility that was able to interpret the structure created by the automatic programmer in order to provide justifications. The thesis of Swartout is that knowledge to create explanations was used to write the program, but is not part of the program itself in ordinary architectures. Therefore one needs to represent the programming process explicitly and keep the resulting decisions to drive justification later. Thus separate representations were used to drive either problem solving or explanation. This architecture provided explanations (and justifications in particular) that were superior to the explanations created from a direct interpretation of a stack of a rule based system. The architecture of XPLAIN is much more complex then the architecture of ordinary rule based systems. The disadvantage of XPLAIN is that the program has never been scaled up to large applications outside its original domain (medical diagnosis). It is debatable whether the automatic programmer approach is feasible for other methods then heuristic classification.

Chandrasekaran proposed to base explanations on his Generic Task architectures [Chandrasekaran et al., 1989b]. Chandrasakeran makes a distinction between the types of explanation that are now also included in the *Common*Kads communication model. They are named type-1, type-2, or type-3 explanations (see section 4.2).

The Generic Task architecture (apart from the Generic Task methodology) supports type 3 explanations directly; "we automatically couch the trace of that system's problem solving behaviour at the architectural level in terms of its control strategy goals (e.g. establish (hypothesis), reject (hypothesis), redesign) ... Because of the fact that the control strategy is implemented explicitly in the GT architecture, it enables us to explain the problem solving strategy (type-3 explanation) [Chandrasekaran et al., 1989b][15]."

Clancey [Clancey, 1983] proposes explanations at a task level. He noted that an expert system typically performs a task at a higher level then than the goal-subgoal level of a rule base ([Clancey, 1983]). In NEOMYCIN the task of the problem solver has been included within the implemented system; and in this was the system is able to refer to this information to produce type-2 explanations.

6.2.3 KADS Models as a base for explanation

In Chapter 5 it is suggested that the expertise model can be used in providing type-1 and type-2 explanations. The strategy of the problem solver is indeed described (until some level of detail) in the model of expertise. These structures can drive an explainer to produce type 2 explanations. There is here a problem of completeness, because in design the algorithms are chosen that actually determine the precise behaviour of the system; therefore detailed accounts of system behaviour can only be given referring to those algorithms. Because of the fact that those algorithms are not included in the expertise model; the cannot be referred to in the explanations.

6.3 Design of Knowledge Acquisition & Stuffing

A knowledge acquisition process can be considered to consist of two sub-tasks: (1) The identification of types of knowledge and identifying the inference mechanisms that play a role in the task (an interpretation model). (2) Filling in the types of knowledge with actual instances of knowledge in the domain (the conceptual model). The latter task has also been called 'knowledge stuffing'.

The idea behind knowledge stuffing is that knowledge has quite a regular appearance, and once found sufficient classes in which this regularity can be fit (the interpretation model); knowledge acquisition stops and knowledge stuffing can start. Several people have tried to support the knowledge stuffing process, because it is a tedious and expensive job when performed without supporting tools. Tedious because you can easily make errors that turn your system into rubbish, expensive because the experts where the knowledge comes form are often busy or required elsewhere.

In the early MYCIN days the EMYCIN shell seemed already good enough to provide a mechanism and a structure for medical reasoning. It was believed that experts could create their own expert systems by simply stuffing their knowledge in the predefined knowledge classed (which are the rules) over which the inference engine (i.e. forward or backward chainer) was to be run. The medical knowledge appeared not to be that regular, and this caused an increased interest all kinds of models of knowledge of which the *Common*Kads approach is also an example.

One of the most advanced approaches currently undertaken is the PROTEGE-II system [Musen et al., 1987] In PROTEGE-II a task is first analysed and a problem solving method is identified for it. The method - task combination requires a certain type of knowledge to be identified and a tool is generated for it to acquire this specific set of relations in the domain. This tool is supposed to be used by a domain expert to do the knowledge stuffing. Similar lines are followed by John McDermott and his colleagues in in MOLE and SALT [Eshelman et al., 1988].

Within the *Common*Kads expertise model the difference between knowledge acquisition and knowledge stuffing is reflected in the domain layer; the domain layer represents both required knowledge structure and the knowledge contents. So together with the inference layer something is known about the knowledge use, the structure and that should be sufficient to stuff the regular knowledge into it. *Common*Kads does not come with advanced tools like PROTEGE or MOLE for knowledge stuffing. The reason for this is that there is a design process taking place between the specification of the knowledge and the representational formalism that is actually going to implement it. Thus the knowledge specification in the domain layer is still to be translated.

The second reason for the lack of knowledge stuffing in *Common*Kads is the fact that the expertise model is not a complete representation of the expertise to be built into the system. It is not necessary complete in its description of inferences and therefore not necessary complete in the types of knowledge required.

In detailed design algorithms are identified for problem solving. Like shown in the Flight scheduling case study at the end of this report these detailed algorithms specify the knowledge actually needed.

Knowledge stuffing is very difficult without any support. The design of task-method specific shells is essential for acquiring and testing large amounts of regularly structured knowledge. Users of the *Common*Kads approach have therefore to include the task of designing such a shell in their application.

Chapter 7

Using an $(\mathrm{ML})^2$ model for design

Peter Terpstra

7.1 Introduction

The subject of this chapter is a case study on the usage of an $(\mathrm{ML})^2$ model [van Harmelen and Balder, 1992] as input for the design process of a KADS type knowledge based system. The case study is taken from a specification of a scheduling task [Balder et al., 1992].

In general the input for design is the model of expertise, the task model and the communication model (see section 3). This chapter only adresses the usage of the model of expertise for design.

The aim of the case study is to investigate the nature of design decisions. Our hypothesis is that in the context of structure-preserving design as laid in chapter 5 it is possible to define precisely (1) what the design decisions are when one operationalises an $(\mathrm{ML})^2$ description and (2) what the criteria are that guide these decisions. This work should lead to a more general *theory* for rationalising design decisions. Such a theory can provide considerable support to the knowledge engineer and can provide the basis for powerful design support tools.

The formulation of such a theory is only possible if one fixes a number of more general design decisions. Chapter 3 makes a distinction between (1) system design where the hardware platform and the target language is chosen, (2) the architecture design, which defines the computational paradigm and (3) the application design, which uses the system and architecture specification to implement the application. In our case study we assume the following choices:

- **System design**: We use the paradigm of logic programming and Prolog as implementation language.
- **Architecture**: Our arcitecture will adopt the KADS distinction between different layers (four layer model) and the $(\mathrm{ML})^2$ notion of dividing a logical layer in several dif-

ferent theories. Our motivation for this architecture is, that in this way the distance between the (ML)^2 specification and the implementation is as small as possible.

- **Application design**: For the high level application (the decomposition paradigm) we will adopt the idea of structure preserving design (chapter 5). ie. the modular structure of the program will be similar to the structure of the model of expertise.

The structure of this chapter is as follows: Section 7.2 gives a general introduction on how to map a formal specification to an implementation in a logical representation language. Section 7.3 introduces the subject of the case study: a scheduling system. Section 7.4 describes the design of the domain layer, section 7.5 describes the task layer and section 7.6 describes the design of the inference layer. The subject of section 7.7 is the communication between inference and domain layer. Finally in section 7.8 we will summarise the general design guidelines which can be generalised out of the case study about using (ML)^2 models for design.

7.2 From specification to implementation

A difference between specification languages like (ML)^2 and (logic) representation languages is that specification languages in general are not designed to be interpreted or executed. Emphasis in specification languages is on expressiveness, declarativeness and not on efficiency or computability. In principle it is possible to build a theorem prover for a formal specification language (eg. Si(ML)^2 [Aben and vanHarmelen, 1992]), but the process of making inferences with such a theorem prover may be undecidable or semidecidable. Therefore one of the possible routes from a formal specification to an operationalised knowledge based system is the translation of the formal specification into another representation for which an efficient computation or interpreter exists.

There are two ways to move from a formal specification of the expertise model towards a working knowledge based system: (1) Stay within the framework of a *declarative language* (like pure functional or logic programming languages). (2) Implement the system in a procedural language by means of *special purpose algorithms* eg. for classification.

7.2.1 Logic based representation languages

Logic based representation languages solve a problem by proving that there are possible instantiations for a query or goal. The proof is constructed by a search through a space consisting of logical consequences of the program. The efficiency of a program is dependent on the size of the search space and of the search control heuristics used by the interpreter. Therefore there are two opportunities to make inferencing in a declarative language relatively efficient: reducing the search space or using optimised search heuristics:

- In general it is the case that the more expressive a declarative language is, the larger the set of of logical consequences of a specified theory are and the more expensive the interpreters for that language are. So one way to move from a specification to a working program is to look first how much expressiveness really is needed and next transform the specification into another specification in a language that is as simple as possible. Some considerations with respect to this trade off between efficiency and expressiveness are

  - Is negation needed? If so, has it to be complete negation or variants like closed world assumption or stratified negation [Ullman, 1988]?
  - Is the universe of discourse finite?
  - Is a representation in Horn clauses possible?
  - Is it possible to write the problem in proposition logic?
  - $(ML)^2$ is a meta layered logic. Is it necessary to represent the system as a meta layered system? One should always try always to reduce the number of interactions between the object meta layers as much as possible.
  - Is recursion needed?

- Another source of inefficiency are the general purpose weak search heuristics which are used by interpreters. Sometimes it is possible to use knowledge about characteristics of a class of similar tasks to construct an efficient special purpose interpreter and internal representation.

  A good example is reasoning about taxonomies. Subsumption relations could be interpreted as a set of universal implications, which are structured around a partial ordering. There are efficient algorithms to traverse partial ordered graphs.

  Another example is a special purpose interpreter for constraint satisfaction [Güsgen, 1987].

7.2.2 Special Purpose Algorithms

A second possibility is to implement the specification by means of special purpose algorithms in an efficient language, eg. an optimised algorithm for classification tasks. Usually these algorithms are implemented in imperative languages. So one has to prove that the algorithms satisfy the formal specification.

7.2.3 Combinations of types of operationalisations

Possible combinations between these routes are

- Procedural attachments.
  One could use a general purpose interpreter for the overall reasoning. Part of the knowledge is defined by special purpose algorithms that efficiently compute some subgoals.

- Use an imperative language for control on the task layer (global control) but use an interpreter for the actual domain layer reasoning (local control) (cf. KARL [Angele et al., 1991], or the algorithm for diagnosis from first principles of Reiter [Reiter, 1987])

7.2.4 Reducing $(ML)^2$

$(ML)^2$ [van Harmelen and Balder, 1992] is a pure specification language where the emphasis is on expressiveness and not on efficiency.

- It uses three logical layers. the task, inference and domain layer.
- The task layer is written in QDL [Harel, 1984]. QDL is a modal logic. This language is undecidable.
- The currently used language for the domain layer and inference layer is full first order predicate logic (semidecidable).
- Most reasoning is done on the inference layer. This means that to maintain the causal connection between inference layer and domain layer a lot of communication overhead is needed between these two layers.

We will split up our analysis in the following case study wrt. the three (ML)^2 layers and the connections between them. For each part we will look for possible language reductions or special purpose interpreters or procedural attachments.

7.3 The Case Study: A Scheduling System

The subject of the case study is a system for simple design. Design in the sense of constructing a complex object (not to be confused with design in the context of software engineering). The object could for example be a physical system, a plan or a schedule.

This system is used as an example for the comparison of different formal specification languages during the ECAI-92 workshop on Formal Specification Methods for Complex Reasoning Tasks [van Langevelde and Treur, 1992]. We will use the (ML)^2 specification of [Balder et al., 1992] which was constructed for that workshop[1].

The above description of the system is independent of the type of object that is constructed. In the workshop the object that is designed was a schedule, consisting of assignments of consumers which consume some resource, to providers, which provide the resource. The inference structure which was constructed for this task in [Balder et al., 1992] is shown in figure 7.1. In the classification of [Puppe, 1990] it is an example of an allocation task: the solution consists of allocation relations between objects of disjunct sets, which satisfy particular requirements. Allocation tasks are a subclass of routine design [Chandrasekaran, 1988b]; a form of design where all parts of the designed object are known beforehand. The claim behind the example scheduler is not that it provides an optimal scheduler, but that it is a simple example suitable for comparing different knowledge level specification languages.

The reasoning pattern in the system is as follows:

1. **Propose**
    (a) **Generate Assignments**: Generate a set of possible parts or configurations of the object
    (b) **Assignment Selection**: Select an assignment wrt. user defined selection heuristics.

---

[1] Due to space constraints our explanation of the (ML)^2 specification will be very short and we refer for details to [Balder et al., 1992].

   (c) **Add Assignment**: Add the assignment to the constructed object (the schedule).

2. **Design Object Evaluation**: Test if the combination of the new assignment and the sofar constructed schedule does not violate user defined requirements.

3. **Test**:

   (a) **Design Process Evaluation**: When the former test detects that there is a violation wrt. user requirements, look for a part of the object which is responsible for the violation: the culprit.

   (b) **Revise Assumption**: Remove the culprit and all object parts which are dependent on it.

4. If the object is finished then stop, else go to 1.

Figure 7.1: The inference layer of the scheduling task

In the example the scheduling is applied on a domain where the consumers are activities and the providers are time periods. So the schedule consists of a set of assigments (allocations) of activities to time periods. The selection heuristics of figure 7.1 are that an assigment A is preferred before another assigment B if A is scheduled before B.

7.4 Domain layer

An $(ML)^2$ domain layer is specified in a many sorted first order predicate logic. The domain theory is divided in a set of modules. A module consists of a import declaration, which defines the modules which are imported, a **signature**, which specificies the language used to describe the **axioms** part of the module.

On the domain layer two sorts are defined: **activity** and **period**. There are four instances of **activity**: **a1**, **a2**, **a3** and **a4**. The sort **period** has three instances **t1**, **t2** and **t3**. These are defined in domainmodule **caseData**

There are only two relations: **at** and **before**. Properties of these relations like transitivity, noncommunitivity and irreflexivity are defined in module **worldKnowledge**:

```
domainmodule worldKnowledge
    import language;
    axioms
        ∀ T1,T2,T3:period (before (T1, T2) ∧ before(T2, T3) → before(T1, T3));
        ∀ T1,T2:period (before(T1, T2) → ¬ (before T2, T1));
        ∀ T1 :period ¬ before(T1, T1);
        ∀ T1,T2:period ∀ A:activity (at(A,T1) ∧ at(A, T2) → T1 = T2));
enddomainmodule
```

The requirements which are used by the inference **evaluateObject** at the inference layer are defined in the module **domainRequirements**.

```
domainmodule domainRequirements
    import caseData;
    axioms
        ∀ T1,T2:period (at(a3,T1) ∧ at(a2, T2) → before(T1, T2));
        ∀ T1,T2:period (at(a3,T1) ∧ at(a4, T2) → before(T1, T2));
        ∀ T1,T2:period (at(a2, T1) ∧ at(a4, T2) → ¬ T1 = T2);
enddomainmodule
```

Finally there is a module **Solutions**, where the final schedule is stored.

7.4.1 Symplifying the representation language

As argued in the former sections typical ways to map a specification on a logical program is to choose a logic as simple as possible ("language reduction") or use special purpose interpreters or to look for procedural attachments. Language reductions can be realised by looking at the characteristics of the implementation language itself and by using the characteristics of the application.

7.4.1.1 Reducing overhead for sort information

$(ML)^2$ uses a many sorted logic. Many sorted logic does not add expressiveness to a language, but often makes inferencing more efficient, because information about sorts reduces the number of possible unifications. Ofcourse in Prolog it is not possible to implement sorts directly, but we could map them on unary predicates.

Inspecting the domain layer and the inference layer in section 7.7 we will notice, that the only place where knowledge about sort information is important for reasoning is during lifting.

Design decision:
On the domain layer the information on sorts is not used for reasoning. They only are used for the definition of lift rules (section 7.7).

7.4.1.2 Model is finite

The second observation we are able to make is that the number of objects on the domain layer is finite because

- there is a finite number of instances of `activities` and `periods`,
- there are no functions on the domain layer,
- there is only place where the *tell* predicate is used, which asserts a solution at the domain layer. This solution is not further used for inferencing.

Because the number of objects in the theory is finite, in principle the theory could be transformed into a propositional theory.

Design decision:
For our design, we use the constraint on the definitions on the domain layer that the number of objects in that module have to be finite. This will guarantee that inferencing on the domain layer is decidable.

7.4.2 Are Horn clauses possible

Because the standard Horn Clause format of Prolog is more efficient than the full first order format of $(ML)^2$ sentences, we will transform the domain layer representation into Horn clause format.

Several axioms are not directly transferable to Horn clauses.

- The predicate before. The noncommunitivity and irreflexivity of `before` are expressed as

  $$\forall\ T_1,T_2\text{:period (before(}T_1,\ T_2) \rightarrow \neg\text{ (before }T_2,\ T_1));$$
  $$\forall\ T_1\text{:period }\neg\text{ (before }T_1,\ T_1);$$

  Assuming Prolog type of SLD resolution, we are able to circumvent this:

```
notBefore(T1,T1).                 % irreflexivity
notBefore(T1,T2):- before(T2,T1). % antisymmetry
before(T1, T3):- T1 \= T3,        % transitivity
                 before(T1, T2),
                 before(T2, T3))).

% An activity can only be assigned to one time period.
equalTime(T1,T2):-at(A,T1), at(A,T2).
equalTime(T1,T1).
```

- Also the constraints as defined in domainRequirements are defined by non Horn clauses, because the head of the implications are negations, eg.

$$\forall\ T_1,T_2{:}\text{period}\ (\text{at}(a_2,T_1) \wedge \text{at}(a_4,T_2) \rightarrow \neg\ T_1 = T_2);$$

A standard trick is to add an auxillary predicate equal-time.

```
% domainRequirements

before(T1,  T2) :- at(a3, T1), at(a2, T2).
before(T1,  T2) :- at(a3, T1), at(a4, T2).
not_equal(T, T) :- at(a2, T), at(a3, T).
not_equal(T1, T2):- not (T1 == T2).
```

7.4.2.1 Domain layer can not directly in Prolog

Unitil now we succeeded to implement all domain layer axioms as Horn clauses. This would suggest to implement the domain layer directly in Prolog. But all inferencing on the domain layer is invoked by the reflective predicates at the inference layer (section 7.7). These inferences need information about the modular structure of the domain layer. Moreover the lifting mechanism need information from the (ML)^2 signatures. Therefore we choose the following representation of the domain layer:

```
%%%------------------------------------------------------------------------
%%%                Language and WorldKnowlege
%%%------------------------------------------------------------------------

domainModule(language).
classes(language,[activity,period]).
relations(language,[at(activity,period), before(period,period),
                    notBefore(period,period)]).

domainModule(worldKnowledge).
import(worldKnowledge,[language]).

axiom(worldKnowledge,(notBefore(T1,T1))).                % irreflexivity
axiom(worldKnowledge,(notBefore(T1,T2):-before(T2,T1))). % antisymmetry
axiom(worldKnowledge,(before(T1, T3) :-  T1 \= T3,       % transitivity
                                         before(T1, T2),
                                         before(T2, T3))).

% An activity can only be assigned to one time period.
axiom(worldKnowledge,(equalTime(T1,T2):-at(A,T1), at(A,T2))).
axiom(worldKnowledge,(equalTime(T1,T1))).

%%%------------------------------------------------------------------------
%%%                Case Data
%%%------------------------------------------------------------------------

domainModule(caseData).
import(caseData, [worldKnowledge]).

instances(caseData,activity,[a3,a1,a2,a4]).
```

```
instances(caseData,period,[t1,t2,t3]).

axiom(caseData, before(t1,t2)).
axiom(caseData, before(t2,t3)).

%%%-----------------------------------------------------------------
%%%               Domain Requirements
%%%-----------------------------------------------------------------

domainModule(domainRequirements).
import(domainRequirements,[caseData]).

axiom(domainRequirements, (before(T1,T2):- at(a3,T1), at(a2,T2))).
axiom(domainRequirements, (before(T1,T2):- at(a3,T1), at(a4,T2))).
axiom(domainRequirements, (not_equal(T1,T2):- at(a1,T1), at(a4,T2))).
axiom(domainRequirements, (not_equal(T1,T2):- not (T1 = T2))).
```

7.5 Task layer

7.5.1 Language Reduction for task layer

When we use Prolog to represent a task layer specified in $(ML)^2$ we map a modal logic (QDL) onto another logic (Prolog). Three ways to map a modal logic program onto Prolog are: (1) Write a meta interpreter in Prolog, which implements QDL logic. (2) Reify the modal logic in a nonmodal logic by providing extra state variable arguments for task layer programs. (3) Make an application dependent language reduction.

We use a combination of the second and the third option because in the scheduling application we do not need the full expressive power of QDL.

Some important differences between QDL and Prolog are

- QDL knows the concept of program state, Prolog does not.
- QDL uses destructive variable assignment. Prolog has only logical variables.
- QDL realises repetition by means of iteration. Pure Prolog uses recursion.
- QDL makes a distinction between programs and predicates. In Prolog every predicate is a program as well.

In $(ML)^2$ there are two different types of global state variables, DKR-variables and PIA history variables. Each PIA has associated with it a variable which contains the history of former invocations of the PIA. DKR-variables contain the previous computed values of intermediate DKR's.

The PIA-state variables are used to define different invocations of PIA's. Such a variable "V_{PIA}" consists of all so far computed input/output pairs of the PIA.

- The predicate has-solution-PIA does not use PIA-state variables.
- The predicates old-solution-PIA(Input,Output) tests if the pair (Input,Output) is a member of V_{PIA}.

- more-solutions-PIA(Input,Output) tests if there is a solution not previously stored in V_{PIA}.
- The QDL program give-solution-PIA finally changes the program state. After a succesfull invocation it stores the input and output arguments in the state variable. Moreover it uses the variable to guarantee that each stored reasoning result is unique.

In principle the state variables could be used for other purposes than for the definition of give-solution-PIA. An example for such usage is to let control be dependent on global properties of the solution set. Another example is the usage of these variables for explanation properties.

Destructive assignments on global variables could be simulated in Prolog by using combinations of assert and retract. But this not only destroys the semantics of Prolog programs, it is also quite inefficient.

A standard way to deal with variables which change during the call to a Prolog clause is to provide two arguments for the variable: an input and an output argument. The changes of the input argument are reflected on the output argument. In combination with the usage of recursion for the implementation of QDL iteration we can avoid destructive assignment.

Design decision:
We do not use global variables on the task layer, but simulate them by extending each tasklayer predicate which was defined in QDL as a program by pairs of logical arguments for each global variable.

Design Decisions	
$(ML)^2$ task layer	Prolog
Destructive Assignment	Input Output Argument Pairs
Iteration	Recursion

7.5.2 Application specific reduction of task layer

The main control loop of the scheduling is specified in $(ML)^2$ as

```
initialise ;
WHILE (¬ test(ObjDescr, finished))
DO propose(ObjDescr, NewObjDescr) ;
    IF test(NewObjDescr, violation)
    THEN revise-task(NewObjDescr, RevisedObjDescr)
    FI
OD ;
tell(solution, ObjDescr)?;
```

initialise $\equiv V_{ObjDescr} := [[\ ]]$

The propose subtask first generates a set of assignments, selects one assignment and adds it to the object description:

```
propose(ObjDescr, NewObjDescr) ≡
    give-solution-generate(Cons,Prov,ObjDescr,Culprits,AssS) ;
```

give-solution-selectAssignment(AssS, Ass) ;
give-solution-add(Ass, ObjDescr,NewObjDescr)

Next the proposed assignments are tested to see if any requirement is violated:

$$\mathsf{test(ObjDescr, Value)} \leftrightarrow \mathsf{has\text{-}solution\text{-}evaluateObject(ObjDescr, Value)};$$

If any requirement has been violated, the assignement set is revised:

revise-task(ObjDescr,RevisedObjDescr) $\equiv$
give-solution-evaluateProcess(ObjDescr,Culprit) ;
give-solution-revise(ObjDescr,Culprit,RevisedObjDescr)

For a full implementation of give-solution-PIA we have to provide two extra arguments for the representation of the PIA-state variable. Moreover we have to add an extra subgoal after the computation of a solution, which tests whether the computed solution was not computed previously (using the input argument for the state variable). In Prolog this test is a bit redundant, because the computations of Prolog are deterministic.

Morover Prolog does not make a distinction between predicates and programs. Each clause can be read as a predicate (declarative semantics) or as a program (procedural semantics).

Design Decision	
$(\mathrm{ML})^2$ task layer	Prolog
Explicit State	Implicit State
Distinction between programs and predicates	Prolog predicates

What we loose in the above language reduction is the concept of an explicit program state, but we do not need it for the scheduling application.

7.5.2.1 Implementation of V_{Dkrs}

The second usage of global variables is for the representation of DKR's. For example the variable Culprits can be changed in the above specification during each cycle in the while loop in reviseTask. This fact is hidden in the above task layer description[2].

Design Decision:
When a DKR variable is only inspected by a PIA, we provide one argument in the PIA predicate. When a DKR variable is also changed by a PIA, we use the convention of design decision **??** and use an input, output argument pair.

Design Decision	
$(\mathrm{ML})^2$ task layer	Prolog
V_{Dkrs} of Input DKR	One argument of PIA predicate
V_{Dkrs} of Output DKR	One argument of PIA predicate
V_{Dkrs} for Input/Output DKR	Two arguments for PIA predicate

---

[2] The convention in software engineering is to avoid the usage of global variables as much as possible. It makes it difficult to inspect programs, because in principle they can be changed and read everywhere in the program. In this respect the $(\mathrm{ML})^2$ conventions are disputable.

7.5.3 Representation for tasklayer

In our design, we replace the while loop by recursion. The **if** ... **then** ... tests are replaced by alternative clauses for schedule-1. Moreover we do more work on initialisation. The reasons for this are explained in section 7.6.2.1.

```
main(Schedule):-
  init_assignment_candidates(AssignmentCandidates),
  schedule(AssignmentCandidates,[],Culprits,Schedule).

% The schedule loop:

schedule(AssignmentCandidates,ObjDescr,Culprits,Schedule) :-
        propose(AssignmentCandidates,ObjDescr,Culprits, NewObjDescr),
        schedule_1(AssignmentCandidates,NewObjDescr,Culprits,Schedule),
        tellAxiom(generate,
                  set_of(function([consumer, provider, assigned])),
                  solution,
                  Schedule).


schedule_1(AssignmentCandidates,ObjDescr,Culprits,Schedule):-
        test(ObjDescr, TestResult),!,
         ( TestResult == finished,  % The stop condition on the while loop.
           Schedule = ObjDescr
        ;  TestResult == violation, % The violation condition in IF....THEN
           revise_task(ObjDescr,
                       AssignmentCandidates,
                       Culprits,
                       NewCulprits,
                       RevisedObjDescr),
           schedule(AssignmentCandidates,
                    RevisedObjDescr,
                    NewCulprits,
                    Schedule)).

% The case when there is no violation and the schedule is not finished:

schedule_1(AssignmentCandidates,ObjDescr,Culprits,Schedule):-
    schedule(AssignmentCandidates,ObjDescr,Culprits,Schedule).
```

Propose is implemented as

```
propose(AssignmentCandidates, ObjDescr, Culprits, NewObjDescr):-
   generate(AssignmentCandidates,ObjDescr,Culprits,Assignments),
   selectAssignment(Assignments,Assignment),
   add(Assignment,ObjDescr,NewObjDescr).
```

Test(ObjDescr,Value)implemented as

```
test(ObjDescr,Value):- evaluate(ObjDescr,Value).
```

Finally ReviseTask is implemented as

```
revise_task(ObjDescr, AssignmentCandidates,
            Culprits, NewCulprits, RevisedObjDescr):-
    evaluateProcess(ObjDescr,Culprit),
    revise(Culprit, ObjDescr, AssignmentCandidates, Culprits,
           NewCulprits, RevisedObjDescr).
```

7.6 Inference layer

7.6.1 Language reduction

An (ML)^2 inference layer is written in a modular many sorted logic enhanced with declarations which define the language. Most of this information is not used for reasoning, but is used for documentation purposes and lifting.

Design decision:

For the inference layer we use an unsorted logic.

Design Decision	
Inference layer	
Many Sorted Logic	Unsorted Logic

The extra information on sorts, which is needed for the communication between inference layer and domain layer (lifting) is hardwired in procedural attachments (section 7.7).

All PIA's are defined as definitions ($\leftrightarrow$). The only way PIA's are used is by calling them as subgoals from the task layer.

Design decision:

We represent PIA's as Horn Clause Rule.

Design Decision	
Inference layer	
Full First Order Logic	Horn Clause Logic
Biimplication	Implication

7.6.2 Application Specific Optimisation

7.6.2.1 Generate

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

The argument objDescr denotes the partial schedule, computed in previous loops. The argument culprits denotes the set of assignments, computed sofar 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 not to be culprits.

```
pia generate
    inputdkrs
        consumers,
        providers,
```

```
        objectDescription,
        culpritsPlusContext;
    outputdkr assignments;
    signature
        predicates
            alreadyAssigned: consumer $\times$assigned set ;
            knownAsViolation: assigned $\times$assigned set ;
        piapredicate $PIA_{generate}$: consumer set $\times$provider set $\times$assigned set $\times$
                                  culpritContext set $\times$assigned set;
    axioms
        $PIA_{generate}$(Consumers, Providers, ObjDescr, Culprits, Assignments) $\leftrightarrow$
            Consumers = $\{$C |$input_{consumers}$(C)$\}$ $\wedge$                              (1)
            Providers = $\{$P |$input_{providers}$(P)$\}$ $\wedge$                              (2)
            $input_{ObjectDescription}$(ObjDescr) $\wedge$                                 (3)
            $input_{CulpritsPlusContext}$(Culprits) $\wedge$                               (4)
            Assignments = $\{$assignment(C, P)|                                  (5)
                          (C $\in$ Consumers $\wedge$ P $\in$ Providers $\wedge$
                          $\neg$ alreadyAssigned(C, ObjDescr) $\wedge$                      (6)
                          $\neg$ knownAsViolation(assignment(C, P), ObjDescr)) $\}$;     (7)

        alreadyAssigned(Consumer,ObjDescr) $\leftrightarrow$
            $\exists$ Provider:provider (assignment(Consumer, Provider) $\in$ ObjDescr);

        knownAsViolation(Assignment, ObjDescr) $\leftrightarrow$
            $\exists$ Culprits : culpritContext set, $\exists$ $ObjDescr_1$ : assigned set
                ($input_{CulpritsPlusContext}$(Culprits) $\wedge$
                $ObjDescr_1$ $\subseteq$ ( ObjDescr $\cup$ $\{$ Assignment $\}$ ) $\wedge$
                culprit($ObjDescr_1$, Assignment) $\in$ Culprits);
endpia
```

The conditions (1) and (2) in the PIA `generate` contain the construct {`C | predicate(C)`}, which is defined as the set consisting of all possible instantiations of the variable `C`, which satisfy predicate.

The predicate $\mathsf{input}_{\mathsf{consumers}}$ is defined as

$$\forall X : \mathsf{consumer}\ (\mathsf{input}_{\mathsf{consumers}}(X) \quad \leftrightarrow \quad \mathsf{ask}^{\in}\ (\mathsf{caseData}, X))$$

The mapping from the domain layer sort instances to the inference sort consumer is defined in the DKR consumers:

```
dkr consumers
    fromdomain caseData;
    use inferenceLanguage;
    dkrvariables A:instance;
    signature
```

```
        constants ⌈A⌉:consumer;
        dkrpredicate input$_{consumers}$:consumer;
    mapping lift(A:activity) ↦ ⌈A⌉;
enddkr
```

In other words the call to ask$^{\in}$ in equation (7.1) instantiates the set of consumers with all lifted activities. The same considerations apply to the set of providers, which is instantiated with all members of the sort period.

If we would take the specification literally as a logical program, then each time generate is called, these sets are to be constructed again. The result will always be the same, because no tell or del predicate is defined on casedata or Worldknowledge, which could introduce or remove terms. As a consequence it is most economical to construct the set only once.

Moreover the only usage of consumers and providers is to generate the set assignments in equation (5). This is the cartesion product of consumers and providers minus already assigned assignments in ObjDescr and the set of violations, defined by the DKR Culprits.

Design decision:
We define a new variable: candidate-assignments. This variable is initialised only once during the initialisation of the task. This design decision seems to be a violation of the structure preserving design paradigm of chapter 5. The assignment candidates are not computed any longer in selectAssignments but during the initialisation of the task.

```
init_assignment_candidates(CandidateAssignments):-
   input_all_consumers(Consumers),
   input_all_providers(Providers),
   findall(assignment(C,P),
          (member(C,Consumers),
           member(P,Providers)),
          CandidateAssignments).
```

The other arguments of PIA generate. objectDescription, assignments and culprits, are intermediate knowledge roles. So we keep them as arguments.

```
generate(CandidateAssignments,ObjDescr, Culprits, Assignments):-
         findall(Ass,
                 (member(Ass,CandidateAssignments),
                 not alreadyAssigned(Ass, ObjDescr),
                 not knownAsViolation(Ass,Culprits,ObjDescr)),
                 Assignments).
```

The predicate knownAsViolation uses the definition of input-CulpritsPlusContext, which in QDL inspects the DKR variable CulpritsPlusContext. So we provide an extra argument for knownAsViolation. When we have a close look on the definition of that predicate, we see that the second condition is a test on subsets of ObjDescr. The third condition is just a test on membership.

design decision:
There will be more subsets of ObjDescr then matching assignments. Therefore we do the cheapest test first. This gives us the following code:

```
knownAsViolation(Assignment,Culprits,ObjDescr):-
     member(culprit(ObjDescr1,Assignment),Culprits),
     subset(ObjDescr1,ObjDescr).
```

7.6.2.2 selectAssignment

selectAssignment selects an assignment from a set of candidate assignments, which is not less preferable than another assignment. Preference between assignments are computed as the domain layer. It is defined as

```
pia selectAssignment
   inputdkrs assignments, selectionHeuristics;
   outputdkr assignment;
   signature
      piapredicate $PIA_{SelectAssignment}$:assigned set $\times$assigned ;
   axioms
      $PIA_{SelectAssignment}(AssS, assignment(A_1, T_1)) \leftrightarrow$
         $input_{assignments}(AssS) \wedge$
         $assignment(A_1,T_1) \in AssS \wedge$
         $\exists A_2$:consumer $\exists T_2$:provider $((assignment(A_2,T_2) \in AssS \wedge \neg A_1 = A_2) \rightarrow$
                                          $\neg ask^{\vdash}$ (caseData,preference$(T_2,T_1)$));
endpia
```

Taking the definition of SelectAssignment literally, would result in a program which randomly chooses an assignment from AssS and compares it with all other members of AssS to check whether there is another assignment which is more preferred than the selected one. If this is the case, again an assignment would be chosen and the comparison would start again. This would in worst case a comparison of all members of AssS with each other. The cost would be quadratic on the length of AssS

A more optimal solution which has a complexity linear to the length of AssS is to use a standard algorithm for computing a maximal element of a set:

```
FUNCTION maximal-element(AssS)
   MaxEl := first element of AssS.
   REPEAT
    El2 = next element of AssS.
    If more-preferred(El2,MaxEl)
    MaxEl := El2
   UNTIL (end of AssS).
   RETURN MaxEl
```

In principle there is an even more optimal solution: Preference criteria for assignment candidates are never changed. Therefore we could sort the set of assignment candidates in initAssignmentCandidates. SelectAssignment would in this case only have to take the first element of assS. So linear time complexity of SelectAssignment could be further reduced to constant time complexity. Actually this is a flaw in the specification of the scheduling task, which does not describe an optimal efficient scheduler.

When we would use this optimisation, the implementation would not directly reflect the implementation and we would have to rewrite the original $(ML)^2$ and CML specifications (structure preserving design). Therefore we do not implement it.

Design decision:
SelectAssignment is implement by algorithm similar as maximal-element.

7.6.2.3 addAssignment

AddAssignment adds a selected assignment to the partial schedule ObjDescr.

```
pia add
    inputdkrs assignment;
    outputdkr objectDescription;
    signature
        piapredicate $PIA_{add}$:assigned $\times$assigned set $\times$assigned set;
    axioms
        $PIA_{add}$(Ass, ObjDescr, $ObjDescr_1$) $\leftrightarrow$
            $input_{assignment}$(Ass) $\wedge$
            $input_{ObjectDescription}$(ObjDescr) $\wedge$
            $ObjDescr_1$ = [ Ass |ObjDescr ];
endpia
```

The implementation is straightforward and directly reflects the $(ML)^2$ definition:

```
% add(Assignment,ObjDescr,ObjDescr1)
add(Assignment,ObjDescr,[Assignment | ObjDescr]).
```

7.6.2.4 evaluateObject

EvaluateObject object checks if the current ObjectDescription does not violate the requirements as defined on the domain layer. It also checks if the schedule is completed.

```
pia evaluateObject
    inputdkrs requirements, objectDescription;
    outputdkr conclusion;
    signature
        piapredicate $PIA_{EvaluateObject}$:assigned set $\times$controlConstant ;
    axioms
        $PIA_{EvaluateObject}$(ObjDescr, violation) $\leftrightarrow$
            $input_{ObjectDescription}$(ObjDescr) $\wedge$
            $ask^{\vdash}$ (domainRequirements, violationObject(ObjDescr));

        $PIA_{EvaluateObject}$(ObjDescr, finished) $\leftrightarrow$
            $input_{ObjectDescription}$(ObjDescr) $\wedge$
            $\neg$ $ask^{\vdash}$ (domainRequirements, violationObject(ObjDescr)) $\wedge$
            $ask^{\vdash}$ (domainRequirements, finishedObject(ObjDescr));
endpia
```

The input DKR requirements is defined as

```
dkr requirements
    fromdomain domainRequirements;
    use objectDescription, conclusion;
    dkrvariables A, T:variable; S:sentence;
```

```
    signature
        functions
            violationObject:assigned set ⟶controlConstant;
            finishedObject:assigned set ⟶controlConstant;
    mapping
        lift( S → ⊥ ) ↦ violationObject(lift(S));
        lift( S → ∀ A:activity ∃ T:time (at(A,T)) ↦ finishedObject(lift(S));
enddkr
```

The domain specific violation criterion in this example task is inconsistency of the current objectDescription with the domain requirements (first rule above). The termination criterion is the assignment of all activities to a given time slot (second rule above).

The two tests of evaluateObject are

```
evaluateObject(ObjDescr,Result):-
  ask_prove(domainRequirements, DomainEvaluation),
  ((DomainEvaluation == violationObject(ObjDescr),
    Result = violation);
   (DomainEvaluation == finishedObject(ObjDescr),
    Result = finished)).
```

7.6.2.5 evaluateProcess

evaluateProcess identifies the responsible assignment of an ObjectDescription, which is known not to satisfy the requirements, by a very simple heuristic. It assumes that the last assignment added to the schedule is the culprit. A violation of the requirements on the schedule is always an assignment in combination with a schedule. Therefore Revise a pair of the responsible assignment plus the ObjectDescription.

```
pia evaluateProcess
    inputdkrs objectDescription;
    outputdkr culpritsPlusContext;
    signature
        piapredicate PIA_EvaluateProcess:assigned set ×culpritContext ;
    axioms
        PIA_EvaluateProcess(ObjDescr, Culprit) ↔
            ∃ MostRecentAssignment:assigned
                (input_ObjectDescription(ObjDescr) ∧
                ObjDescr = [ MostRecentAssignment |_ ] ∧
                Culprit = culprit(ObjDescr, MostRecentAssignment));
endpia
```

Again the implementation is trivial:

```
evaluateProcess([Culprit|ObjDescr], culprit([Culprit|ObjDescr],Culprit)).
```

7.6.2.6 Revise

The final PIA states how to remove the culprit, generated by evaluateProcess according to the revision heuristics. The implementation is straight forward.

```
pia revise
    inputdkrs objectDescription, culpritsPlusContext, revisionHeuristics;
    outputdkr objectDescription;
    signature
        piapredicate $PIA_{revise}$:assigned set $\times$assigned $\times$assigned set ;
    axioms
        $PIA_{revise}(ObjDescr, Culprit, ObjDescr_1) \leftrightarrow$
            $\exists$ Context : assigned set $\exists$ C : assigned
                ($input_{ObjectDescription}(ObjDescr) \wedge$
                $input_{CulpritsPlusContext}([Culprit \mid \_ ]) \wedge$
                Culprit = culprit(Context, C) $\wedge$
                ObjDescr = (ObjDescr$_1$ $\cup$ C)) ;
endpia
```

```
revise(CulpritPlusContext, [Culprit |RevisedObjDescription],
       Culprits, [CulpritPlusContext|Culprits], RevisedObjDescription):-
        CulpritPlusContext = culprit(_ObjDescr,Culprit).
```

7.7 Communication between inference layer and domain layer

$(ML)^2$ defines the KADS relation between domain and inference layer as an object meta relation. The inference layer is a meta theory, which reasons about the domain layer theory. To be able to stay in the framework of first order logic, the access from the inference layer to the domain layer is indirect via a naming relation. Domain layer constants, variables and functions are named by inference layer constants. Domain layer sentences named by inference layer functions. This mapping between domain layer constructs and inference layer constructs is realised by a term rewriting grammar consisting of lift rules.

The advantages of maintaining this indirect communication between domain and inference layer are that it enlarges the reusability of knowledge bases and provides information for explanation facilities. On the inference layer reusability is enhanced because specifications of inference structures can be written relatively independent of particular domain theories. On the domain layer, theories are reusable because they can be used for more than one inference structure and domain axioms could be applied in one inference layer theory by different inferences in different ways.

Moreover, because the lift rules specify the roles of domain layer constructs in inferencing they contain important information for explanation facilities.

$(ML)^2$ defines the connection between inference and domain layer by means of knowledge roles (section 7.7.1) and reflective predicates (section 7.7.5).

7.7.1 Knowledge roles

7.7.2 Format for lift mappings

In $(ML)^2$ the specification of a lift rule is of the form:

lift(DomainLayerTemplate ↦ InferenceLayerTemplate)

For example: lift(T:period) ↦ ⌈T⌉

The domain layer template contains sort all necesarry sort information. The sort information for the inference layer template is speficied in the signatures of the DKR. For example in the DKR providers below, the DKR variable T is defined as being of the type instance and the constant ⌈T⌉ is of the sort provider.

```
dkr providers
    use inferenceLanguage;
    dkrvariables T:instance;
    signature
        constants ⌈T⌉:provider;
        dkrpredicate input_{providers}:provider ;
    mapping lift(T:period) ↦ ⌈T⌉;
enddkr
```

Design decision:
We move most of this information to the rule itself. This will result in a rule of the form

```
     mapping(DKR, lift(DomainLayerSort, DomainLayerTemplate,
                  InferenceLayerSort, InferenceLayerTemplate))}

For example:
     mapping(providers,lift(instance(period), T,
                            constant(provider), nameOf(T))).
```

7.7.2.1 Lift Rule Parser

A reflective predicate is always called in a PIA or DKR. The first thing that has to be done when a reflective predicate makes a call to the domain layer is to transform its inference level argument into a domain layer argument by means of an applicable liftrule. This rule has to be found in DKR modules which are used (imported) by the PIA or DKR in which the reflective predicate is called. The applicability of a lift rule is defined by the type of its inference layer template, which has to match with the corresponding argument of the reflective predicate.

When an applicable rule is found, a parser maps the inference layer template of the rule on an initial domain layer template. This may invoke other lift rules. For each invoked lift rule there will be on the domain layer a search for the type declarations through (imported) domain layer signatures.

The algorithm is as follows:

```
GETMAPPING(DKrs, DomLayerSort, InfLayerSort, FromDomain, InferenceLayerTerm)
  Find DKR with same FromDomain
  FOR each rule in DKR
     IF  the rule  matches with the DomLayerSort and InfLayerSort
     THEN parse(DKR, DomLayerSort, InfLayerSort, FromDomain, InferenceLayerTerm)
  END FOR
```

Algorithm `Parse` handles the different types of lift rules. In the scheduling application there are three cases:

1. The rule is of the form:

   (lift Domain constants $\mapsto$ Inference layer constants).

   For example

   lift(T:period) $\mapsto$ $\lceil$T$\rceil$

   Parsing these types of rules is straight forward:

   (a) Find the domain layer module where the constants of that sort are defined.
   (b) Instantiate the inference layer constants with the quoted domain layer constants.

2. The domain layer type is a relation and inference layer type is a function with lifted arguments. The rule is of the form:

   lift(Atomic domain sentences -> Inference layer functions)

   For example:

   lift(at(A:activity,T:period)) $\mapsto$ assignment(lift(A), lift(T))

   This is realised by the algorithm:

   (a) Find the domain layer module where the relation is defined.
   (b) Apply the renaming as defined in the liftrule:
   For each argument:

       call getMapping(DKR, FromType(Arg),FromTemplate (Arg),
       ToType(ARG) ,ToTemplate (ARG).

   (c) Return the instantiated ToTemplate.

3. The domain layer type is a conjunction and inference layer type is a set. For example:

   lift(at(A:activity,T:period) $\wedge$ Others) $\mapsto$ [lift(at(A:activity,T:period)) |lift(Others)];
   lift($\top$) $\mapsto$ [ ];

   The parsing realised by the algorithm:

   (a) Lift the first conjunct to the first element of the set.
   (b) Lift the rest of the conjuncts to the rest of the set.
   (c) Return the instantiated templates.

DesignDecision:
The quoted domain layer constants as used in case 1 of Parse make the output of the parser less readable. Therefore we decide the following refinement of Parse:
We do not use quote domain layer constants. We can safely do this as long as the set of domain layer constants is disjunct of the inference layer constants, which is indeed the case in the scheduling application.

7.7.3 Optimisation of naming relation

A drawback of the indirect mapping via lift rules is that it introduces a large computational overhead needed to implement the communication between the two layers. Each query constructed at the inference layer, involves applying a term rewriting where lift rules and knowledge bases have to be parsed and new goals have to be formulated.

Our design goal is to construct a mechanism which reduces this computational overhead, but preserves the explicit information about knowledge roles as defined in the lift rules. Therefore this section explores some possibilities for optimisation of the lifting mechanism.

The main sources for the computational overhead during the parsing of rewrite rules are (1) the search for inference layer modules in which the liftrule is defined, (2) the matching through type hierarchies. (3) the recursive nature of the lift rules. (4) and on the domain layer the search through modules imported by the Fromdomain and (5) the matching between domain layer type hierarchies.

7.7.3.1 Compiling the structure of the knowledge base

The structure of the knowledge base is fixed, so the DKR's which contain applicable rules are always the same. Also the set of domain layers where matching domain constructs can be found is fixed. If we specialise each lift mapping, which is called from a reflective predicate in a DKR or PIA into a mapping with extra arguments, which denote the name of the PIA or DKR and the applicable domain layer module, then we avoid the search through the structure of the knowledge base.

This gives a rule format of the form:

```
        mapping(PIA,ActualFromDomain,
                lift(FromType, Template, ToType, ToTemplate)).
```

For example

```
        mapping(providers,
                lift(constant(period),T, constant(provider), nameOf(T))).
```

will be compiled in the PIA selectAssignments as

```
        mapping(selectAssignments,caseData,
                lift(constant(period),T, constant(provider), nameOf(T))).
```

7.7.3.2 Compilation of Sort Hierarchies

Both inference layer terms and domain layer expressions use a many sorted logic. Two terms unify if they are of an equal sort, or if one of the sorts is a subsort of the other sort. The resulting term has a sort, which is the most specific of the two original sorts.

Unification in a sort hierarchy is computational more expensive than unsorted unification. From the $(\text{ML})^2$ specification it can be deduced, what the actual sort will be of the term, which will be matched with the lift rule template. Therefore a rule, which is specialised for a PIA can be further optimised by specialising its domain layer and inference layer types to the corresponding types as used in the matching domain and inference layer constructs.

7.7.3.3 Removing recursion in grammar

Some rules are recursive. The template in the right hand side, invokes a new application of lifting. A rule like:

$$\mathsf{lift(at(A{:}activity,T{:}period)} \mapsto \mathsf{assignment(lift(A),\ lift(T));}$$

will invoke two extra calls to `getmapping`. From the $(\text{ML})^2$ specifications it could be deduced which rules are applicable for $\mathsf{lift(A)}$, and $\mathsf{lift(T))}$. So this rule could be rewritten into

$$\mathsf{lift(at(A{:}activity,T{:}period)} \mapsto \mathsf{assignment(}\lceil \mathsf{A} \rceil,\ \lceil \mathsf{T} \rceil\mathsf{);.}$$

In rules which apply on sets like:

$$\mathsf{lift(at(A{:}activity,T{:}period)} \wedge \mathsf{Others} \mapsto \mathsf{[lift(at(A{:}activity,T{:}period))\ |lift(Others)]);}$$

it is not obvious beforehand how many conjuncts the conjunction will contain. This is application dependent. A solution for this is to rewrite the rule into a set of specialised rules for 1 .. N different elements of the sets.
For example the specialised rule for two conjuncts would look like

$$\begin{array}{l}\mathsf{lift(at(A1{:}activity,T1{:}period)} \wedge \mathsf{at(A2{:}activity,T2{:}period)} \mapsto \\ \quad \mathsf{[assignment(}\lceil \mathsf{A1} \rceil,\ \lceil \mathsf{T1} \rceil\mathsf{),\ assignment(}\lceil \mathsf{A2} \rceil,\ \lceil \mathsf{T2} \rceil\mathsf{)])}\end{array}$$

Finally the original rule would be maintained as a catch all for sets with a length greater than N.

7.7.3.4 Optimisation of Parser

The last possible optimisation would be to compile the parser itself. Given a parser written in pure Prolog, it can be specialised for each invocation of a specific lift mapping.

7.7.4 Discussion

We haved proposed several optimisations for the indirect mapping between domain and inference layer. Optimisation is realised by specialising the lift rules wrt. the structure of the knowledge base, the sort hierarchies and the recursive calls in lift rules and/or specialising the parser itself.

These proposals are generic and do not change the semantics of an $(\text{ML})^2$ model. Moreover the optimisations can be automised and be supported be tools. With exception of the optimisation of the parser itself, it is possible to do the optimisations on the original $(\text{ML})^2$ specification. This has the advantage that we need only one optimisation tool for different computational environments , which use a mechanism for the communication between de domain and inference layer, which is similar to the one as designed for the scheduling application.

The optimisation techniques can be implemented by using a combination of partial evaluation (PE) and explanation based generalisation (EBG) [Mitchell et al., 1986, van Harmelen and Bundy, 1988]. PE compiles a function by solving a partial uninstantiated goal in all possible ways. The result of a PE compilation is a specialisation of the function wrt. the instantiated arguments. Such a specialisation consists of a set of conditions in terms of ground facts.

A problem with PE is that there are no general solutions to deal with recursive subgoals. With uninstantiated arguments it is unclear where to stop the evaluation of recursive predicates. Therefore we use the concept of an operationality criterium as used in explanation based generalisation. An EBG compilation stops when it meets a goal, which is defined as being operational. By changing the operationality condition we can optimise the naming relation to different levels of generality. The choice of the level of generality is dependent on the type of knowledge based system:

1. Optimise only the inference structure. This makes it possible to optimise components of libraries of generic specifications.
2. Optimisation (1) plus optimisation of generic domain layer partitions (as partition worldKnowledge in the scheduling application.
3. Optimisations (1) and (2) plus signatures for case data.
4. Optimise the whole knowledge base.

Optimisations (1), (2) and (3) would result in a lifting protocol very similar as the solution in chapter 5, where the lift rules specify a direct mapping between domain layer sorts and inference layer sorts.

7.7.4.1 Caching

Finally there is yet another optimisation of a different nature. This optimisation is specific for systems with a finite domain as is the case the scheduling application, where the sets of both inference layer and domain layer constructs are finite. Therefore it makes sense to use caching as an optimisation technique.

Each inference layer term, which is defined by the lift mechanism corresponds to one unique domain layer construct [van Harmelen, 1992]. So the naming relation is a partial function from inference layer to domain layer. Therefore in finite domains it is possible to construct a lookup table with inference layer constants as keys, consisting of entrys as `(inference-layer-constant,domain-layer-term)` and fill this table during reasoning.

7.7.5 Reflective Predicates

In the scheduling application three different reflective predicates are used: ask^L , $\mathsf{ask}^\vdash$ and tell. Tell is a straightforward assertion of axioms in a module on the domain layer. ask^L (Module, Expression) tests whether there is a language element of Expression defined in Module. This can be realised by just applying the lift mechanism.

7.7.5.1 Ask-prove

$\mathsf{ask}^\vdash$ (Module, InferenceLayerTerm) lifts InferenceLayerTerm downwards to DomainLayerGoal and calls the object layer interpreter to solve it. $\mathsf{ask}^\vdash$ is used in the pia selectAssignment and in evaluateObject. In selectAssignment the to be proven domain layer goal is straightforward and can directly be solved by a backward chaining Horn clause meta interpreter [Sterling and Shapiro, 1986].

The invocation of $\text{ask}^\vdash$ in evaluateObject is much more complicated.

$$\begin{aligned}&\mathsf{PIA}_{\mathsf{EvaluateObject}}(\mathsf{ObjDescr,\ violation}) \leftrightarrow\\ &\quad \mathsf{input}_{\mathsf{ObjectDescription}}(\mathsf{ObjDescr}) \wedge\\ &\quad \mathsf{ask}^{\vdash}\ (\mathsf{domainRequirements,\ violationObject(ObjDescr)});\end{aligned}$$

It uses the mapping

$$\mathsf{lift(\ S} \rightarrow \perp\ ) \mapsto \mathsf{violationObject(lift(S));}$$

The goal which is constructed for violationObject is a test on inconsistency of a set of hypothetical assigments wrt. the domain requirements. This goes beyond the capacities of a simple Horn clause backward chaining interpreter. There are two ways to deal with this problem (1) refine the $(ML)^2$ specification or (2) build a special purpose interpreter

7.7.5.2 Refining the $(ML)^2$ specification

The sentences in domainRequirements are implications of the form

$$\forall\ \mathsf{T_1,T_2{:}period\ (at(a_3,T_1)} \wedge \mathsf{at(a_2,\ T_2)} \rightarrow \mathsf{before(T_1,\ T_2))}$$

A set of assignments is consistent with a requirement of this form if

- the requirement is not applicable, ie. not all the conditions of the requirement are provable given the set of assignments. Or stated more operational for the type of requirements as used in the scheduling application: not all conditions of the requirement unify with members of the set assignments.
- or if the requirement is applicable and the conclusion of the requirement is provable on the domain layer.

Therefore we specialise the $(ML)^2$ specification as follows:
We lift the requirements to an inference layer function requirement: assigned set x constraint by means of a mapping

$$\mathsf{lift(Conditions} \rightarrow \mathsf{Constraint)} \mapsto \mathsf{requirement(lift(Conditions),\ lift(Constraint))}$$

We define a new predicate $\mathsf{input}_{\mathsf{requirements}}$:

$$\begin{aligned}&\forall\ \mathsf{X{:}requirement\ (input_{requirements}(X)} \leftrightarrow\\ &\qquad\qquad \mathsf{ask}^{\in}\ (\mathsf{domainRequirements,requirement(Conditions,Constraint)))}\end{aligned}$$

Next we redefine the PIA-predicate EvaluateObject (for conciseness we omit sort information):

$$\begin{aligned}&\mathsf{PIA}_{\mathsf{EvaluateObject}}(\mathsf{ObjDescr,\ violation}) \leftrightarrow\\ &\quad \exists\ \mathsf{Requirements,\ Conditions,\ Constraint}\\ &\qquad \mathsf{Requirements} = \{\mathsf{R}\ |\mathsf{input_{requirements}(R)}\} \wedge\\ &\qquad \mathsf{requirement(Conditions\ ,Constraint)} \in \mathsf{Requirements} \wedge\\ &\qquad \mathsf{Conditions} \subseteq \mathsf{ObjDescr)} \wedge\\ &\qquad \neg\ \mathsf{ask}^{\vdash}\ (\mathsf{caseData\ ,Constraint});\end{aligned}$$

7.7.5.3 Special purpose interpreter

The former specialisation made a refinement of the $(\mathrm{ML})^2$ model itself. The effect of the refinement is that the standard backward chaining Horn clause interpreter can be used for the invocations of $\mathrm{ask}^{\vdash}$. Another possibility is to make a special purpose interpreter, which knows what axioms are to be used as requirement and how to apply a requirement. This involves less lifting, because the special treatment of domain layer clauses is hardwired in the interpeter and does not have to be specified anylonger at the inference layer. The interpreter is complete and consistent for these special cases of requirements as used in the scheduling example:

```
checkRequirement(DomainModule,Assumptions,Requirement):-
     Requirement = (Conclusion :- Conditions),
     subset(Conditions, Assumptions),
     domainSolve(DomainModule,Conclusion). % use the standard meta interpreter.
```

DesignDecision:
For testing assumptions wrt. requirements we use the following special purpose interpreter, which uses conditions of requirement clauses not as normal conditions, but as applicability tests: is the constraint in the head of the clause applicable for the set of assumptions.

7.8 Conclusions

The subject of this chapter was a case study on the design of a knowledge base system using an $(\mathrm{ML})^2$ specification as input. Wrt. the design models in chapter 2 we have taken the following decisions:

- **High Level Architecture**:
  We use a KADS type architecture which is similar as the one defined in figure 5.2 in chapter 5, which supports structure preserving design.

  For this case study we restricted ourselves to the inference process. No facilities for communication with the outside world are provided.

- **Detailed Architecture**:
  We have chosen the following knowledge representations and interpreters:
  - Task layer:
    The task layer is directly implemented in full Prolog. The only non logical control construct, that we needed are cuts. Subtasks are implemented as Prolog procedures.
  - Communication between task layer and inference layer is implemented as subgoaling.
  - Inference layer:
    PIA's are represented by Horn clauses. The interpreter for PIA's is pure Prolog plus negation by failure. We did not have to use control constructs like cuts. The $(\mathrm{ML})^2$ many sorted logic is reduced to unsorted logic.
  - Communication between inference layer and domain layer:
    The communication is indirect via lift rules and reflective predicates. DKR's

  are implemented as a set of Prolog facts and interpreted by a parser. This is the only place where sort information is used.

  Reflective predicates are designed as meta interpreters. We needed two interpreters, a standard SLD backward chaining interpreter and an interpreter for checking consistency of assumptions with requirements.

  - Domain layer:
    The domain layer is implemented as a modular Horn clause logic. This is the only layer not represented in Prolog itself, because the meta interpreters which are called by reflective predicates need information on modules.

- **System Design**:
  Prolog is used as well as representation language as programming language for building parsers and interpreters.

- **Application design**:
  The chapter gave a detailed account on issues, which come up when trying to map $(ML)^2$ on Horn clause logic.

Most of the solutions of the reduction of $(ML)^2$ to the chosen representation are generic, in the sense that they are applicable for similar architectures and knowledge representations:

- The mapping of the $(ML)^2$ representation of the task layer in QDL on Horn clauses.
- The mapping of PIA's, in $(ML)^2$ defined as bidirectional implications, on Horn clauses.
- The design and suggested optimisations for the algorithms, which realise the communication between the inference and domain layer.

7.8.1 Relation with KADS architecture

If we compare the solutions as found in this chapter with the components of the architecture of figure 5.2 then we see that the architecture developed for the scheduling application is a specialisation of the generic KADS architecture.

- The working memory in the architecture for the scheduling application is implemented by providing arguments for task layer and inference layer predicates. It is not a separate module.
- The task interpreter is the Prolog interpreter itself.
- Inference functions are designed as Horn clause predicates.
- Inference methods are designed as meta interpreters, called by reflective predicates.
- KBS access functions are implemented by a parser, which parses lift rules. In section 7.7.3 suggestions were given on how to compile the lift rule grammar into a more direct mapping from inference layer sorts to domain layer sorts as is used in chapter 5.

7.8.2 Design processes for the different design models

7.8.2.1 Design process for high level architecture design

In section **??** we gave some guidelines on how to map an (ML)^2 specification on a more operational logical knowledge representation. Our heuristics for doing this are

- Choose a logic as less expressive as possible,
- Look for opportunities for using interpreters specialised for certain reasoning tasks.
- Supplement the system with procedural attachments when needed.

In this application we only had to use the first two options. It was not necessary to use procedural attachments.

7.8.2.2 Design process for detailed architecture design

The design process for the detailed architecture is of a different nature than the application design process. Detailed architectures specify the knowledge representation language and interpreters. This is very similar as the design in "traditional" knowledge representation. One has to specify the symbol level descriptions of the data structures and interpreters. Appropriate specification languages are Z [Spivey, 1988] or VDM [Jones, 1986b]. The difference of (ML)^2 with these specification languages is that the latter are used for the specification of abstract data types. (ML)^2 is a language for the specification of models.

Normally a Z or VDM specification starts with a very high level specification. The design process is a cycle of specialisations. In each design cycle the abstract specification is refined by adding more (implementation oriented) details. This continues until the specification is so detailed that the mapping of the