KADS[II]

ESPRIT Project P5248 KADS-II

An Advanced and Comprehensive Methodology
for Integrated KBS Development

# The CommonKADS Library

Agnar Aamodt *
Bert Bredeweg †
Joost Breuker †
Cuno Duursma *
Christiane Löckenhoff ‡
Klas Orsvarn §
Jan Top ¶
André Valente †
Walter Van de Velde *

* *Vrije Universiteit Brussel*
† *University of Amsterdam*
‡ *Siemens AG*
§ *Swedish Institute of Computer Science*
¶ *Netherlands Energy Research Foundation*

| | |
|---|---|
| **Document Id.** | KADS-II/T1.3/VUB/TR/005/1.0 |
| **CEC Reference** | Draft Deliverable D1.3a, Version 1 |
| **Date** | May 4, 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

# The CommonKADS Library

Agnar Aamodt *, Bert Bredeweg †, Joost Breuker †, Cuno Duursma *, Christiane Löckenhoff ‡, Klas Orsvarn §, Jan Top ¶, André Valente †, Walter Van de Velde *

* *Vrije Universiteit Brussel*
† *University of Amsterdam*
‡ *Siemens AG*
§ *Swedish Institute of Computer Science*
¶ *Netherlands Energy Research Foundation*

**Synopsis**

This document summarizes the results to date of Task I.3. in the ESPRIT project KADS-II (P5248) "Development of generic task and domain models". The goal of this task is the specification of an initial library of modelling components that can serve as the building blocks for complete expertise models. The major achievements that are being described are (i) definition of the nature of the library entries, (ii) definition of a language to describe them, (iii) definition of a library structure, (iv) initial population of the library, and (v) analysis of applications and their impact on the library.

| | |
|---|---|
| **Document Id.** | KADS-II/T1.3/VUB/TR/005/1.0 |
| **CEC Reference** | Draft Deliverable D1.3a, Version 1 |
| **Date** | May 4, 1992 |
| **Distribution** | Consortium |

*Revision date:* April 3, 1992

*Author:* Walter Van de Velde

*Modifications:* Compilation of a first vesion of the deliverable from the following contributions:

- "The Common KADS Library" (KADS-II/T1.3/WP/UvA, Bert Bredeweg, Joost Breuker, André Valente, April 1, 1992)
- "The CommonKADS model library" (KADS-II/T1.3/VUB/RR/004/0.1, Walter Van de Velde, March 23, 1992)
- "Definition of the CML" (KADS-II/TI.3.1/WP/VUB/006/1.0, Cuno Duursma, Walter Van de Velde and Agnar Aamodt, 23 September 1991

The main aim of this compilation has been to merge the views present in the different documents. Extensive editing of the contributions has been avoided as much as possible. However, different parts of a single document are not necessarilly kept together. A preface was added (part of the introduction from UvA document) and a new introduction is based on introductions from UvA and VUB contributions. Authorship of the resulting document is for all authors together. The views expressed in this version are not yet approved by all partners and incoherency remains.

*Revision date:* April 4, 1992

*Author:* Walter Van de Velde

*Modifications:* Re-organization of the deliverable. Chapter 2 of UvA input (Basic Strategic Move) has been extensively edited and rewritten to reflect the main lines of integration between the different views. It was renamed to "Overal design of the library". Planning of further revision plan.

*Revision date:* April 7, 1992

*Author:* Walter Van de Velde, Agnar Aamodt

*Modifications:* Preface, Introduction and Design chapter revised after feedback from Agnar Aamodt. First restructuring of other chapters from UvA input. Separation of parts on "Basic Modelling Components", "Generic Models", "Library use" and "Library Contents". Attempt to ensure coherent terminology throughout the document (e.g., Kads-I terminology replaced).

*Revision date:* April 8, 1992

*Author:* Walter Van de Velde

*Modifications:* Addition of Appendices B, C, and so forth to cover the library contents. These additions are based on the following contributions:

- "Role
  Limiting methods for the CM library" (KADS-II/T1.3/VUB/XXX/0.1, Cuno Duursma, March 20, 1992)
- "Input for the Library" (KADS-II/T1.3.6/WP/SIEM/002/0.1, Christiane Löckenhoff, March 6, 1992)
- "Towards realistic CML components for Top Down Fault-Localization" (KADS-II/T1.3/WP/SICS/003/0.1, Klas Orsvärn, February 24, 1992)
- "Input Based on Problem Solving Methods and Task Types form CONSTRUCT" (KADS-II/T1.3/WP/VUB/004/0.1, Agnar Aamodt, March 20, 1992)

In addition two parts were extracted from Chapter 5 of the UvA input, namely a part on Canonical Inference Functions and a part on Basic Tasks. The Appendices have hardly been edited, except for an introductory paragraph that situates them within the framework set up in the main part of the deliverable. A full integration of the library contents description has not been done, since the framework has not been completely approved by the project. Authorship of the appendices is with the respective authors.

*Revision date:* April 9, 1992

*Author:* Walter Van de Velde

*Modifications:* Chapter 3 (CML) based on UvA contribution, but structure slightly changed and summary added. Also added clarification on relation to D1.1 models. Chapter 4 changed title to "Basic Modeling Components". No significant changes. Chapter 5 changed title to "Generic Models". Most of original UvA input has become appendix. Now based mainly on VUB input. Upate of the conclusion.
Chapters on usage and approaches have not been updated. They are probably no longer consistent with the present version.
This version send out to UvA for feedback.

*Revision date:* April 14, 1992

*Author:* Walter Van de Velde

*Modifications:* New version of appendix on fault-localization (version by Klas of April 10th
Revision and insertion of Chapter on usage based on UvA material (their Chapters 6 (Use of the library) and 7 (Realizing Different Modelling Approaches)
Revision of all chapters based on feedback from UvA.
Addition of introductory section in Appendices that situates them better in the library framework (Appendices need further integration).
The version of April 14th is submitted for internal review and delivered to the partners at the plenary meeting in Garmisch.

.

*Revision date:* April 29, 1992

*Author:* Walter Van de Velde

*Modifications:* Update of entire document based on internal review reports from the following people:

- Santiage Gala (Eritel)
- Rob Taylor (Touche Ross)
- Frank Van Harmelen (UvA)

In addition comments and the results from various discussions at the plenary meeting in Garmisch (April 21-24) were incorporated. Joost Breuker, Manfred Aben, Hans Akkermans and in particular Jan Top provided valuable comments on the draft.
The major goal of this update round is to improve consistency, shorten the document, and clarify various issues.

*Revision date:* May 3, 1992

*Author:* Walter Van de Velde, Agnar Aamodt

*Modifications:* Preparation of review document version 1.0
New appendices versions from Klas Orsvarn, Christiane Löckenhoff and Jan Top integrated.
Extensive updating and streamlining. Making explicit points of contention, especially with respect to D1.1.
A number of more or less drastic changes were required to arrive at a coherent theory (the editor takes full responsability for these). In particular Section 2.2.3 has been extensively updated (focussing more on operators than before), the notion of feature has been removed entirely from Chapter 4, the composite methods are moved from Chapter 4 to Chapter 5.

*Revision date:* July 10, 1992

*Author:* Walter Van de Velde

*Modifications:* Minor changes to main part based on comments from Jan Top and Andre Valente.
Appendix on BNF replaced by new version.

# Preface

This report is the intermediate result of work on Task I.3. within the KADS-II project (P5248). The objective of the Task I.3 is to deliver a library of *generic modeling components* that can be used to support a knowledge engineer in the development of a model of expertise. In this report we present the basic design for the CommonKADS library, together with examples of the entities that can be stored in it.

This deliverable is the result of the confrontation - not without problems - of sometimes strongly opposing views from various partners in the task. Basically three lines of work have been contributing to the final result. At VUB (Agnar Aamodt, Cuno Duursma, Walter Van de Velde) the library design was approached as being largely independent of D1.1 work. The notion of "generic model" was taken as primary and the task was to design a structure that helps to store and use a large variety of such generic models. The internal structure of the models, however, was considered to be in the realm of DI.1 work and reflected only in the library through the nature of "modeling operators". A second line of work, at UvA (Bert Bredeweg, Joost Breuker, Andre Valente), has focussed on developing an interpretation of the modeling framework presented in DI.1 [Wielinga et al., 1992a][1] that could be the basis for the structure of the library. The main output of this effort is an interpretation and - where necessary - an expansion of the conceptual modeling framework in terms of basic modeling components, and a derivation of library structure from model structure. We show in this deliverable (main part) that these two views are not incompatible and, in fact, complementary. In parallel to the two lines of work on the library design, a third line has concentrated on providing contents for the library (Agnar Aamodt, Joost Breuker, Cuno Duursma, Christiane Lurz, Klas Orsvarn, Jan Top). This has resulted in sometimes very thorough analyses of expertise within the framework of DI.1. It was rapidly clear that the application of the conceptual results to these "real" models was far from a trivial task, and more work is required on these models to "enter" them into the library framework (see the various Appendices).

We believe that, in spite of these differences and difficulties, the proposed library design is not a compromise but an elegant and flexible solution to a variety of difficult problems. We are fully aware of the fact that additional work is needed to really present the results of this task in a unified and unambiguous language and framework. This deliverable is therefore an intermediate report on ongoing work, presenting the foundations on which a large and useful CommonKADS model library can be build.

[1] The latest version at our disposal dates from February 6, 1992 (version 3).

# Contents


**1 Introduction** **1**
1.1 Overview . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 2

**2 Overall Design of the Library** **5**
2.1 Issues in the design of the library . . . . . . . . . . . . . . . . . . . . . . . . . . 5
2.2 Basic Design Decisions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7
2.2.1 The elements of the library . . . . . . . . . . . . . . . . . . . . . . . . . 7
2.2.2 The language of the library . . . . . . . . . . . . . . . . . . . . . . . . . 8
2.2.3 The structure of the library . . . . . . . . . . . . . . . . . . . . . . . . . 9
2.3 Support tools . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 10
2.4 Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 12

**3 Practical Interpretation of the Common KADS Modeling Language** **13**
3.1 The CommonKADS Expertise Model . . . . . . . . . . . . . . . . . . . . . . . . 13
3.1.1 Domain Knowledge . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 13
3.1.2 Inference Knowledge . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 14
3.1.3 Task Knowledge . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 15
3.1.4 Mapping between Knowledge Categories . . . . . . . . . . . . . . . . . . 16
3.1.5 Problem Solving Knowledge . . . . . . . . . . . . . . . . . . . . . . . . . 17
3.2 The CML versus Basic Modeling Components . . . . . . . . . . . . . . . . . . . 18
3.2.1 Separating Data Flow, Decomposition and Control . . . . . . . . . . . . 18
3.2.2 Integration of Problem Solving Methods . . . . . . . . . . . . . . . . . . 19
3.3 Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 20

**4 Basic Modeling Components** **23**
4.1 Inference Functions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 23
4.2 Inference Structures . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 24
4.3 Control structures . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 25
4.4 Problem Solving Methods . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 25
4.5 Domain Knowledge . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 27
4.6 Typologies of Basic Modeling Components . . . . . . . . . . . . . . . . . . . . . 27
4.7 Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 28

**5 Generic Models** **29**
5.1 The form of a generic model . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 29
5.2 Modeling operations as library structure . . . . . . . . . . . . . . . . . . . . . . 31
5.3 Classes of Generic Models . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 32
5.4 Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 34

**6 Use of the Library** **35**
6.1 Accesses and Routes . . . . . 35
6.2 Task Features and Component Features . . . . . 35
6.3 Operations on the Expertise Model . . . . . 37
6.4 How to Decide upon an Operation . . . . . 37
6.5 Entering the Library: Basic Tasks and Fields . . . . . 39
6.5.1 Task Division routes . . . . . 39
6.5.2 Domain Division routes . . . . . 40
6.6 How to Specify a Modeling Approach . . . . . 41
6.7 Summary . . . . . 44

**7 Conclusion and Future Work** **45**

**A BNF of the CommonKADS Library** **47**
A.1 Basic Library Components . . . . . 47
A.1.1 Some initial definitions . . . . . 48
A.1.2 Four-part structure of the basic components . . . . . 48
A.1.3 Conceptual description of the components . . . . . 50
A.1.3.1 Inference function . . . . . 50
A.1.3.2 Control structure . . . . . 50
A.1.3.3 Expansion method . . . . . 50
A.1.3.4 Control method . . . . . 51
A.1.3.5 (Role) Refinement method . . . . . 51
A.1.3.6 Inference method . . . . . 51
A.1.3.7 Domain Ontology . . . . . 51
A.1.3.8 Domain Model . . . . . 52
A.1.3.9 Model Schema . . . . . 53

**B Canonical Inference Functions** **55**
B.1 Introduction . . . . . 55
B.2 A classification of inferences . . . . . 57

**C Basic Tasks** **63**
C.1 Introduction . . . . . 63
C.2 Analysis Tasks . . . . . 64
C.3 Synthesis Tasks . . . . . 65
C.4 Modification Tasks . . . . . 66

**D Role Limiting Methods** **69**
D.1 Introduction . . . . . 69
D.2 Cover & differentiate . . . . . 70
D.2.1 Domain Knowledge . . . . . 70
D.2.2 Tasks . . . . . 73
D.2.3 Inference structure . . . . . 74
D.2.4 Dataflow of C&D . . . . . 75
D.2.5 C&D applied on Diagnosis . . . . . 75
D.2.6 C&D applied on Classification . . . . . 76
D.2.7 C&D applied on Case based Assessment . . . . . 77
D.2.8 C&D applied on Design . . . . . 78
D.3 Propose & revise . . . . . 78
D.3.1 Domain Knowledge . . . . . 78
D.3.2 Tasks . . . . . 80
D.3.3 Inferences . . . . . 81

D.3.4 Dataflow of P&R . . . 81
D.3.5 P&R applied on Configuration . . . 82
D.3.6 P&R applied on Scheduling . . . 82
D.3.7 P&R applied on Diagnosis . . . 83
D.4 Extrapolate from a similar case . . . 83
D.4.1 Domain knowledge . . . 83
D.4.2 Tasks . . . 85
D.4.3 Inference structures . . . 86
D.4.4 Data flow . . . 86
D.4.5 ESC applied on Sizing . . . 88
D.4.6 ESC applied on Diagnosis . . . 88
D.4.7 ESC applied on Classification . . . 88
D.4.8 ESC applied on Assessment . . . 89
D.4.9 ESC applied on Design . . . 89
D.5 Comparison . . . 90
D.6 Modelling Problems . . . 90

**E Library Components - Methods and Tasks from CONSTRUCT** **91**
E.1 Introduction . . . 91
E.2 The COMMET Framework . . . 92
E.3 Problem Solving Methods . . . 93
E.3.1 List of Methods . . . 93
E.3.2 Combined methods . . . 95
E.3.3 Descriptions of Problem Solving Methods . . . 96
E.3.3.1 A Very General Method . . . 96
E.3.3.2 A Rather Specific Method . . . 99
E.4 A list of Task types . . . 102

**F Technical Diagnosis by Top-Down Fault-Localization** **105**
F.1 Relation to the library framework . . . 105
F.2 Introduction . . . 106
F.3 The modelling language . . . 106
F.4 Initial version of TDFL . . . 107
F.5 Rational Justification . . . 109
F.6 Features of the HP application . . . 110
F.6.1 Requirements entailed by these task features . . . 110
F.7 Reducing the time required for observations . . . 110
F.8 Decomposition model tailored to diagnosis . . . 111
F.9 Detecting characteristics of the fault . . . 112
F.9.1 Fault as the actual faulty behaviour . . . 112
F.10 Restricting diagnosis to a subset of faulty behaviour . . . 114
F.11 Testing more likely hypotheses first . . . 115
F.12 Hypotheses that are not directly testable . . . 116
F.12.1 Generating auxiliary hypotheses for elimination . . . 117
F.13 Conclusions . . . 118
F.14 Future work . . . 119

**G Reverse Engineering of QUAESTOR with KADS** **121**
G.1 Introduction . . . 121
G.2 Domain Layer . . . 122
G.3 Inference Layer . . . 123
G.3.1 Input and Output . . . 123

G.3.2 Description of dynamic knowledge roles . . . . . . . . . . . . . . . . . . . . 123
G.3.3 Description of primitive inference actions . . . . . . . . . . . . . . . . . . . 124
G.4 Task Layer . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 125
G.5 Reverse Engineering . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 125
G.6 Conclusion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 127

**H Analysis of Assessment tasks 129**
H.1 The Task of Assessment . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 129
H.1.1 The Case–based Approach . . . . . . . . . . . . . . . . . . . . . . . . . . . . 130
H.1.2 The Theoretical Approach . . . . . . . . . . . . . . . . . . . . . . . . . . . . 131
H.2 The Task Decompose . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 132
H.3 The Primitive Inference Decompose . . . . . . . . . . . . . . . . . . . . . . . . . . 133

**I Analysis of Assessment tasks 135**
I.1 Background of the Inference Structure . . . . . . . . . . . . . . . . . . . . . . . . 135
I.1.1 Inference Layer . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 136
I.1.2 Inference Structure . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 136
I.1.3 Input and Output . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 137
I.1.4 Description of Metaclasses . . . . . . . . . . . . . . . . . . . . . . . . . . . . 138
I.1.5 Description of Knowledge Sources . . . . . . . . . . . . . . . . . . . . . . . . 140
I.2 Task Layer . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 140
References . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 144

# List of Figures


2.1 The library contains two categories of elements: basic modeling components and generic models. The basic modeling components are building blocks of expertise models. The generic models are templates for complete expertise models. These two categories of library elements play complementary roles in modeling. (The layout of the library in the figure is not significant.) . . . 8

2.2 Library elements may be described at a verbal, conceptual and formal level. The conceptual level is the central one. Task features serve as starting points of paths leading into the library. . . . 9

2.3 The library is structured in various ways to support its role of retrieving models and guiding the modeling process. The different structuring principles support access into the library (task features), organize the two categories of library elements (component features, typologies and modeling operators) and link them to each other (compositional links). (The layout of the library in the figure is not significant here.) . . . 11

3.1 An expertise model is a large componential structure (part-of) of knowledge items from predefined knowledge categories. For each category DI.1. specifies (potentially) a type-hierarchy for the legal representatives of this category. The process of modelling can be viewed as filling this expertise model frame by pushing down the category representatives along the branches of their typologies until they are sufficiently specific. . . . 14

3.2 Basic modeling components define a fixed level of granularity for knowledge components to be meaningful library entities. Moreover, type hierarchies of different knowledge categories are merged by extending them upwards in order to account for the undetermined type of knowledge items in the library. . . . 19

4.1 Structure of the library: basic modeling components . . . 24

5.1 The library consists of partial (but in form complete) expertise models described at various layers of precision and organized in useful partial model development graphs. 29

5.2 The structure of the library supports modeling by providing a set of useful partial model development graphs. These document the relation between generic models that can be obtained from one another by the application of modeling operators. . 31

6.1 Typology of library component features . . . 36

6.2 Types of operations on the library . . . 38

B.1 Part-of hierarchy of CML domain terms to define types of CIS . . . 56

B.2 Expressions as propositions of CML domain terms . . . 57

B.3 *A schematic view on type (2) and type (3) inferences.* . . . 58

B.4 Typology of Canonical Inference Functions . . . 61

C.1 Classification of synthesis tasks according to what is given. . . . . . . . . . . . . . 65
C.2 Taxonomy of Basic Tasks. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 67

D.1 Inference structure of Cover and differentiate at first level of decomposition. . . . . 74
D.2 Data flow of Cover and differentiate. . . . . . . . . . . . . . . . . . . . . . . . . . . 76
D.3 Inference structure of Propose and Revise . . . . . . . . . . . . . . . . . . . . . . . . 81
D.4 Data flow of Propose and revise. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 82
D.5 Inference structure for extrapolate from a similar case at the first level of decomposition. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 86
D.6 Dataflow of extrapolate from a similar case . . . . . . . . . . . . . . . . . . . . . . 87

E.1 The three perspectives . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 92
E.2 Structures and roles within the three perspectives . . . . . . . . . . . . . . . . . . . 93
E.3 Inference structure for the generic Divide-and-Conquer method. Note that an additional structure (dashed arrows) is introduced in the lower right part of the figure, in order to link the two separate inference structure related to inferences before and after the application of the subtask solving PSM. All arrows can be viewed as input/output flow of information related to the problem case. . . . . . . . . . . . . 100

F.1 The initial version of task bodies used in TDFL. . . . . . . . . . . . . . . . . . . . 108
F.2 The task body of the task establish. . . . . . . . . . . . . . . . . . . . . . . . . . . 113
F.3 The task body of troubleshooting adapted to hypothesis as fault. . . . . . . . . . . 113
F.4 The task body of diagnose with the actual behaviour as fault . . . . . . . . . . . . 114
F.5 The task body of decompose that restricts the hypotheses to a subset of the behaviour. 114
F.6 The task body of decompose based on selecting the most likely hypothesis, and the task bodies of its subtask preferInSituation . . . . . . . . . . . . . . . . . . . . . 115
F.7 The task body of diagnose with a refinement sequence of behaviours as fault . . . 116
F.8 Four components in a signal chain . . . . . . . . . . . . . . . . . . . . . . . . . . . 116
F.9 The task body of the task generateElimination . . . . . . . . . . . . . . . . . . . . 117
F.10 The diagnose task body adapted to handle hypotheses that are not directly testable. 118

I.1 Inference Structure for *assessment* . . . . . . . . . . . . . . . . . . . . . . . . . . 137
I.2 Inference Structure for *find term* . . . . . . . . . . . . . . . . . . . . . . . . . . . 138
I.3 Inference Structure for *assess term* . . . . . . . . . . . . . . . . . . . . . . . . . . 140
I.4 Inference Structure for *find subterms* . . . . . . . . . . . . . . . . . . . . . . . . . 143
I.5 Inference Structure for *assess subterm* . . . . . . . . . . . . . . . . . . . . . . . . 144
I.6 Criteria Diagram for a (Ficticious) Technology . . . . . . . . . . . . . . . . . . . . 145
I.7 Decomposition Tree of the Task Assessment . . . . . . . . . . . . . . . . . . . . . . 146

# Chapter 1

# Introduction

In this report we present a structure for the CommonKADS library together with examples of models that can be stored in it. The primary role of the CommonKADS library is to support a knowledge engineer in the construction of an expertise model for a given application. Such an expertise model is but one of a series of models that are being developed in a CommonKADS project[Martil et al., 1992]. For knowledge based systems development it is, however, a central one. Providing effective support in constructing an expertise model is therefore a major aim of the CommonKADS methodology.[1]

A secondary role of the CommonKADS library is to integrate the experience on modeling from the wider KBS community. Indeed, the idea of a library of modeling components is not unique to CommonKADS. Several researchers base their work on the hypothesis that similar applications share similar models [?; McDermott, 1988; ?; Steels, 1990]. Achieving a degree of re-usability of models is therefore a common goal of research in knowledge engineering. Different views are around on how this can be realized, i.e., on what should be the nature of the re-usable components and how the modeling process itself can be best characterized. As far as we know ours is the first comprehensive attempt to integrate these different views into a single library.

In light of this integration effort the structure, contents and usage of the CommonKADS library must take into account a variety of insights from the knowledge engineering community. In particular the experiences from previous KADS users (see also [Bauer, 1992]) should be considered. For example the notion of inference structure is generally regarded as an important means to a first understanding of the expertise that has to be conceptualized. At the same time it is acknowledged that inference structures hardly ever completely fit, but instead need modification and further refinement. It is interesting to see how modeling approaches other than KADS have lead to similar insights. For example generic tasks [?] hardly ever completely fit a problem either and they have to be refined and changed in all but trivial cases. Another lesson learned from using KADS is that it is useful to create, during the modeling process, alternative views on the model under construction. In particular, an inference (or data-flow) oriented and a control oriented perspective turn out to be important. With regard to this, the Components of Expertise framework [Steels, 1990] also advocates the maintenance of different views: that of domain models, of task models and of problem solving methods.

So, experiences from practice in modeling are often similar, even though they come from different modeling approaches. It is the unified CommonKADS framework for describing expertise models [Wielinga et al., 1992a] that provides the foundation for accommodating a variety of models into a single library. The structure and contents of the library, however, takes these ideas one step further such that alternative approaches to the process of modeling are facilitated. The present

[1] A similar library could exist for any of the other models in the CommonKADS model suite. Many of the considerations in this report carry over to those libraries. Since the expertise model is so central and distinguishes KBS projects from others, it is natural for the project to focus on these.

document shows how we are approaching this integration task.

The current library is a partial one and it will need to be expanded and maintained throughout the project and afterwards. This is no surprise, given the fact that the CommonKADS methodology presupposes no focus on a particular type of target application. The number of domains and tasks to be covered is therefore enormous. This task is clearly too big to be finished within the scope of the KADS-II project. The main contribution of this report for a knowledge engineer is, therefore, not so much in the detailed library contents that it describes, but rather in making clear what - and what not - to expect from it and how to use it. Further work on the library will focus on filling in the library contents. This and - most importantly - the practical usage of the library will provide the ultimate validation of the proposals that we here put forward.

## 1.1 Overview

The contents of this report is as follows:

- Chapter 2 discusses a number of basic issues relevant for the design of the CommonKADS library. It then presents the overall design of the library with respect to library contents, library language and library structure.
- Chapter 3 discusses the Conceptual Modeling Language (CML) as proposed in Deliverable 1.1 [Wielinga et al., 1992a]. The first part presents a summary of the CML entities as it currently stands. The second part discusses problems and choices that are relevant for deciding upon how the different elements from the CML must be represented in the library in order to be of use for supporting the model construction process.
- Chapter 4 presents the set of basic modeling components that must be defined in the library as well as how they are structurally related. Results in this chapter are based on a re-interpretation of the CommonKADS modeling framework, taking into account the pragmatic requirements posed by the library.
- Chapter 5 discusses generic models. Generic models serve as frames or templates in the construction of complete expertise models. They are complementary to the basic modeling components that can be thought of as providing the contents of the models. Generic models are essential for facilitating alternative approaches to supporting the model construction process.
- Chapter 6 describes how the contents of the library can be accessed by a knowledge engineer. In particular, this chapter discusses different routes of navigation through the library and explains how alternative approaches to supporting the knowledge engineer can be realized.
- Chapter 7 concludes the report by summarizing the major achievements of the work presented in this report. In particular, this chapter points out some differences between the current approach and the previous KADS-I framework.

Following the main part of this report there are several appendixes:

- Appendix A is a proposal for a BNF syntax for describing the entities of the library. This proposal is largely based on [Schreiber, 1992].

The other appendices describe different contents of the library. They are currently only moderately integrated in the overall library design because it makes only sense to do so after the structure and the definition of generic modeling components has been agreed upon. In particular the BNF proposed in Appendix A is not being used yet. Nevertheless, the contents of these appendices is dense and it will require only a reasonable effort to extract from them actual elements to be stored in the CommonKADS model library (see also the introductory section in each of the appendices).

- Appendix B describes the typology of canonical inference functions.
- Appendix C describes the taxonomy of basic tasks.
- Appendix D describes role limiting methods, a class of generic models.
- Appendix E describes problem solving methods and associated task types from the ESPRIT project CONSTRUCT.
- Appendix F describes steps toward library components for top-down fault localization.
- Appendix G identifies inference and task structures by analysing an existing system for selection and configuration of mathematical models.
- Appendix H analyses assessment tasks, and shows how library entities can be extracted from this analyses.
- Appendix I presents an inference structure for technology assessment.

# Chapter 2

# Overall Design of the Library

This chapter is composed of three parts. In the first part, we discuss the basic issues in the design of the library of expertise models. In the second part, we present the overall design of the CommonKADS library and show how it addresses these issues. The third part expands on preliminary ideas toward support tools for maintaining and using the library.

## 2.1 Issues in the design of the library

The first question pertaining to the design of the library is:

*What is being stored in the library?*

There are at least two extreme answers to this. First one could store only complete and ready-made expertise models. Such models would be highly specific for an application class and in principle easy to use, but reasonable coverage would require an enormous amount of such models to be stored. The other extreme is to store only the smallest building blocks of expertise models (knowledge roles, domain entities, task descriptions, control statements, and so forth), as if there were a "shelf" in the library for any type of model element that has been defined in [Wielinga et al., 1992a]. This approach is unlikely to provide much support for a knowledge engineer who would probably get lost in the large number of primitives that can be selected and the huge number of possible combinations that can be made.

It is clear that in fixing the nature of library elements several points have to be taken into account. Care has to be taken that the grainsize is not too small so that components can be meaningfully indexed. On the other hand a larger grainsize typically diminishes the chance that the component be re-usable. Another factor is the degree of guidance in modeling. Smaller components require a lot of modeling still to be done but in themselves don't say anything about how to plan this. This is particularly problematic for expertise modeling since the relative interaction hypothesis [Wielinga et al., 1992a] predicts that there can be any degree of dependency between different parts of a model. So, whatever grainsize one fixes on the library, an additional element of guidance in model construction seems to be needed. In other words, genericity arrises not only from the re-usability of components, but also - if not mainly - from the re-usability of combinations of them.

A second question to be answered is:

*In what language are library elements described?*

The library must cover a wide range of potential users and of syntactic, semantic, formal and predictive restrictions. To meet these requirements it must allow for various levels of precision and different views according to the user's perspective. There are at least three possible ways to describe expertise models: a *verbal*, a *conceptual* and a *formal* description (from the less structured and precise to the more structured one). It seems useful to store any of those in the library as alternative descriptions to be used depending on the state of modeling, the level of precision needed, or the kind of user. However, it would be problematic to allow descriptions in any of these levels indiscriminately – e .g., to allow an element to be described in a verbal way without a conceptual or formal description. In this way, one runs the risk of having items with essentially the same knowledge content stored as different or unconnected elements, because there is no obligatory link between them. Therefore, it is necessary to choose one of these levels as central, in such a way that the description at the other levels are connected to the description at the central level.

Another question to be considered in this respect is whether the CML as proposed in [Wielinga et al., 1992a] is appropriate as is for the library. The library language must of course be consistent with the CML from DI.1 but it must also fit with the practice of modeling. For example, the incremental construction of large models requires facilities for typing, naming and easy referencing. But there are additional elements that will require an extension of the CML and that will be discussed in more detail in Chapter 3.

Another issue to be considered in the library design is:

*How is the library used?*

The primary role of the library is to assist in building an expertise model for a concrete application. For this it is not sufficient for the library to simply store elements, but it must also support finding the most useful ones. Usefulness is determined in part by the various epistemological and pragmatic constraints imposed by the application [Steels, 1991a]. These are the so-called task features[1]. For example, a diagnostic application for systems the behavior of which is only qualitatively known and can be only partially observed will require a considerably different model than when "perfect" knowledge is available. Usefulness is also determined by the restrictions that follow from previous modeling choices. For example, when a problem solving method has been selected a certain type of domain model may be necessary. In summary then the library must be indexed by task features so that informed selection of model elements is facilitated and useful combinations indicated, and so that constraints that follow from modeling decisions are adequately reflected in the alternatives that can be retrieved from the library.

An additional point will turn out to be important. The practice of modeling show that it is not always possible in advance to decide upon the role of a library element in a finished model. A typical example is to choose between the status of "task" or "inference". While this may be clear in the final model, it is not initially. It follows that the library can not follow too strictly the category and type structure of a model of expertise as defined in [Wielinga et al., 1992a]. Certain knowledge elements must have a common "shelf" in the library (e.g. a "task-or-inference") that can reflect the undetermined role status of certain library elements.

A final issue in the design of the library is the following:

*How are different modeling approaches integrated?*

As described in the introduction, a second role of the library is the integration of different modeling approaches [Wielinga et al., 1992a; pp.34-36]. For example the inference structure driven approach of KADS-I [Breuker et al., 1987b] is different from the method driven approach advocated by McDermott [McDermott, 1988]. A modeling approach implies a choice of a type of entry point in the library (e .g., a generic task, an inference structure or a problem solving method) and a paradigmatic approach to modeling (e.g., from inference structure down to domain modeling and up to the task modeling). In order to support these different approaches the library must allow

[1] Task features are primarily studied in Task I.5.

for different access routes into the library. When we view the construction of a complete expertise model as a search process through a space of (possible) expertise models[2], then a second role of the library is to point at interesting steps to be taken in the modeling process. For example, from a given state of the expertise model several refinements may be possible, or the combination of two models into a single new one may be an interesting step to take and the relation between the generic models involved should be documented in the library.

Apart from these four "academic" questions, pragmatic considerations must play an important role in the design of the library. In particular, practical experiences and preferences of KADS users must be taken into account (see also [Bauer, 1992]). The following observations can be pointed out:

- Inference structures are generally regarded as an important means to a first understanding of the expertise that has to be conceptualized, independent from the possible control structures that can be defined over it.
- Inference structures hardly ever completely fit, but instead need modification and further refinement. This is facilitated by the meshing of task and inferences (and therefore also of task decomposition and inference structure) in the library.
- It is useful to have, during the knowledge acquisition process, the possibility of creating alternative views on the contents of the expertise model being constructed. The library explicitly supports a data-dependency oriented and a control oriented perspective.

The experience of KADS users is relevant because it provides experimental data with respect to the practical usefulness of specific aspects of the CommonKADS methodology that the library should support.

## 2.2 Basic Design Decisions

In this section we present our basic decisions related to the design of the library. In particular, we discuss proposals for library elements, for the language of the library and for the structure of the library.

### 2.2.1 The elements of the library

We distinguish between two (disjoint) categories of library elements: the *basic modeling components* and the *generic models* (Figure 2.1).

- The *basic modeling components* are the smallest ones in the library. They are canonical (or *orthogonal*) in the sense that they are one partition of the expertise model (i.e., their contents do not overlap) in elements small enough to provide the desired flexibility by their recombination. This implies a *canonical structure* for every complete model through the necessary relationships between the different types of basic modeling components. For example, an inference function is decomposed into an inference structure, and a control structure is imposed on an inference structure.
- The *generic models* are frames that represent a class of complete expertise models. They capture useful combinations of basic modeling components and serve as templates for the construction of complete expertise models. A generic model is an expertise model frame, complete in form but not necessarily in contents. This notion of generic model provides for any kind of model to support a modeling approach. It can express, for example, Chandrasekaran's generic task as a useful combination of a certain type of task model, domain models, inference structure and control structure.

[2] This is the view being taken in Task I.5 that studies task features

The basic modeling components are inference functions, inference structures, control structures, domain models, case models, domain ontologies and problem solving methods. They will be described in detail in Chapter 4. Basic modeling components provide a useful level of granularity and sufficient flexibility in modeling. In themselves, however, they do not provide any guidance on what are useful combinations. This guidance is contained in the generic models. The notion of generic model is a straight generalization of Chandrasekaran's generic tasks [?]. Just like a generic task a generic model supports modeling because it documents useful ways for bringing together basic modeling components into complete expertise models. The basic structure of a generic model is defined by the canonical structure among basic modeling components. This canonical structure is in a sense the most "empty" generic model.

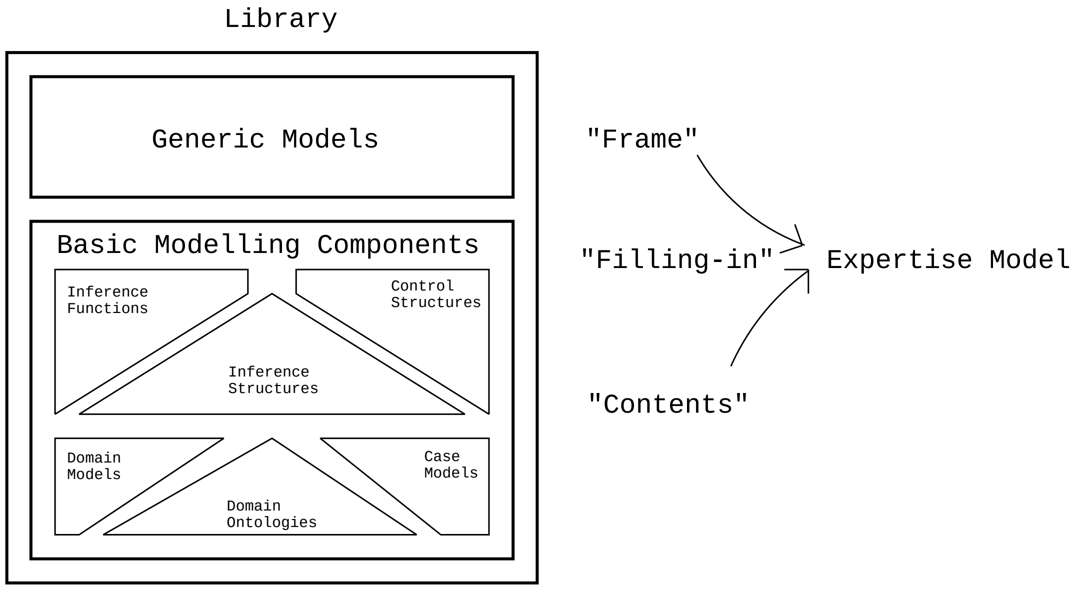



Figure 2.1: The library contains two categories of elements: basic modeling components and generic models. The basic modeling components are building blocks of expertise models. The generic models are templates for complete expertise models. These two categories of library elements play complementary roles in modeling. (The layout of the library in the figure is not significant.)

An interesting aspect to comment in this respect is the question of *redundancy*. While redundancy is extremely undesirable in the category of basic modeling components, this is not true for the generic models. To allow flexibility and the use of different modeling methods and strategies, the various generic models may overlap. For instance, an inference structure may appear both as a separate component and as part of one or several generic components like a Chandrasekaran's generic task.

### 2.2.2 The language of the library

Every library entity (basic modeling component or generic model) can be described as follows (see also Figure 2.2):

1. **Verbal description:** a (partial) knowledge level model of an application in natural language. This is a description in terms of "texts", "quotes" (sequences of words from a text) and "words", and references among these, described by texts themselves. This is like a hypertext language, and some of Shelley's analysis features may be still useful here.

2. **Conceptual description:** a knowledge level model of an application in the "core" CML language for describing knowledge items and structures. The language used is based on an interpretation and extension of the CML [Schreiber, 1992] and is being described in Chapter 3.

3. **Formal description:** a knowledge level model of an aspect of the task reality in the formal language $ML^2$. We refer to [van Harmelen and Balder, 1992] for an introduction to $(ML)^2$ and to [ten Teije et al., 1991] for a more detailed discussion of some of the technical issues.

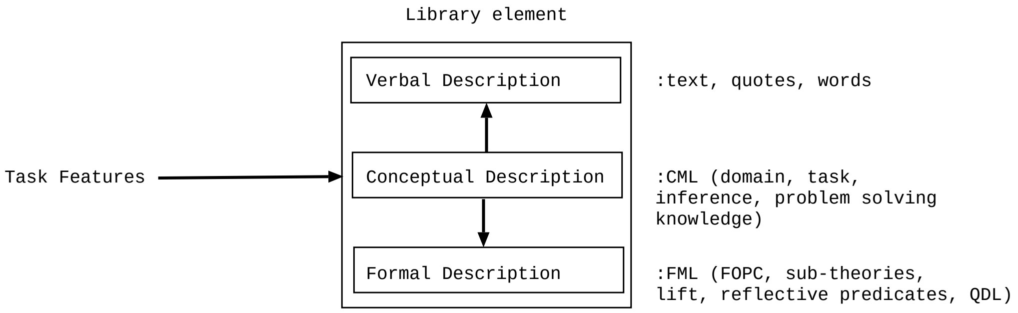



Figure 2.2: Library elements may be described at a verbal, conceptual and formal level. The conceptual level is the central one. Task features serve as starting points of paths leading into the library.

The most adequate level to tie the three levels in a coherent way is the conceptual level. To use the verbal level would still allow confusion because it has no structure. To use the formal one would require a level of understanding of formal issues that is unrealistic to assume for the average knowledge engineer. The conceptual level is both structured enough to avoid lack of integration and informal enough to be understandable to the average knowledge engineer. Thus, library elements are primarily conceptual components with formal and/or verbal descriptions attached.[3]

The conceptual language must be consistent with the CML from DI.1 [Wielinga et al., 1992a] but it must also fit with the practice of modeling using the basic modeling components and generic models (Section 2.2.1). For example, parts of expertise models may be re-usable (and therefore worth while to store in the library) but it may be hard to pre-specify their role in the final model. This is for example the case with tasks and inferences which are sometimes hard to pin down in the initial phases of modeling. In Chapter 3, we will discuss this issue in more detail and propose a useful interpretation and extension of the CML to accomplish these goals.

### 2.2.3 The structure of the library

The library supports two activities: selection of useful model components and generic models, and progression through useful steps in modeling. This is achieved by a number of structuring principles within the library (see also Figure 2.3):

- **Structure among basic modeling components:**
  the sets of basic modeling components are structured in basically two ways:
  - **Typologies of basic modeling components:**
    basic modeling components within a single class are organized in a series of typologies. Typologies facilitate the construction of an expertise model from a generic model by indicating the refinements and alternatives that the library provides. Typologies can be based on various principles, for example on ontological choices (see Appendix B)

[3] The conceptual descriptions are the main focus of attention in this document.

- **Constraints from component features:**
  component features document compatibility constraints between basic modeling component. A component feature can indicate component necessity or component exclusion. For example, the choice of an inference may require or exclude the usage of a certain type of domain model. Component features restrict the set of potential combinations of components to those which are structurally possible.

- **Structure between generic models and basic modeling components:**
  a compositional link relates generic models and basic modeling components. The generic model is a frame which restricts combinations of basic modeling components, typically by type restrictions.[4] The compositional link documents the structure of a generic model in terms of its constituent component types.

- **Structure among generic models from modeling operators:**
  the process of modeling that the library should support can be viewed as a search among expertise models in which various modeling operators are applied to the model under construction.[5] Modeling operators are typically parameterized by library elements and exploit the structures among basic modeling components. For example, a specialization operator may exploit a typology of domain models. When one generic model can be obtained from another one in this way, then this is also documented as a relation between them in the library of generic models. To navigate through the library of generic models is to explore potential modeling episodes.

- **Task features as primary guidance in modeling:**
  a task feature is any characteristic of an application and its environment that informs the construction of an epistemologically and pragmatically adequate expertise model for it. For example, task features can help in the selection of model components or generic model. More generally task features indicate modeling operations (and model selection is one of them) parametrized by library elements. Thus, the task features are the primary indices for accessing, navigating though and exploiting the library in modeling.

Structuring the library of generic models according to modeling operators is a powerful means of decoupling the library structure from the structure of the models being stored. In the library of basic modeling components, modeling proceeds based on fine grained operations that are guided by the structural relationships between the different components. In the library of generic models on the other hand, models that are very different, and could be obtained only by a number of construction steps over components can be directly related to one another. For example, the application of a method to a task typically requires to adjust terminology (see Appendix D). The relation between a generic model that represents a task - method combination is therefore more than simply compositional. Moreover, modeling operations allow for expressing genuine combinations of complete models, like the integration of a classificatory problem solver and a constructive one into a new model [Van de Velde, 1988]. Such operations are not easily expressed using basic components.

## 2.3 Support tools

Several of the tools which have been defined within the project will make use of the library in one way or the other. A particular tool - the *Library Management Tool* will provide the environment for defining new library components and modifying existing ones. This is the tool of the 'librarian', i.e. the system administrator. Tools for the knowledge engineer which have strong relations to the library include the Conceptual Model Editor (CMED), and the task feature tool - referred to

[4] One can envisage different constraints to be represented by the generic model, but before we have seen examples of this we will apply occam's razor to keep things simple.

[5] This view is extensively explored in Task I.5.

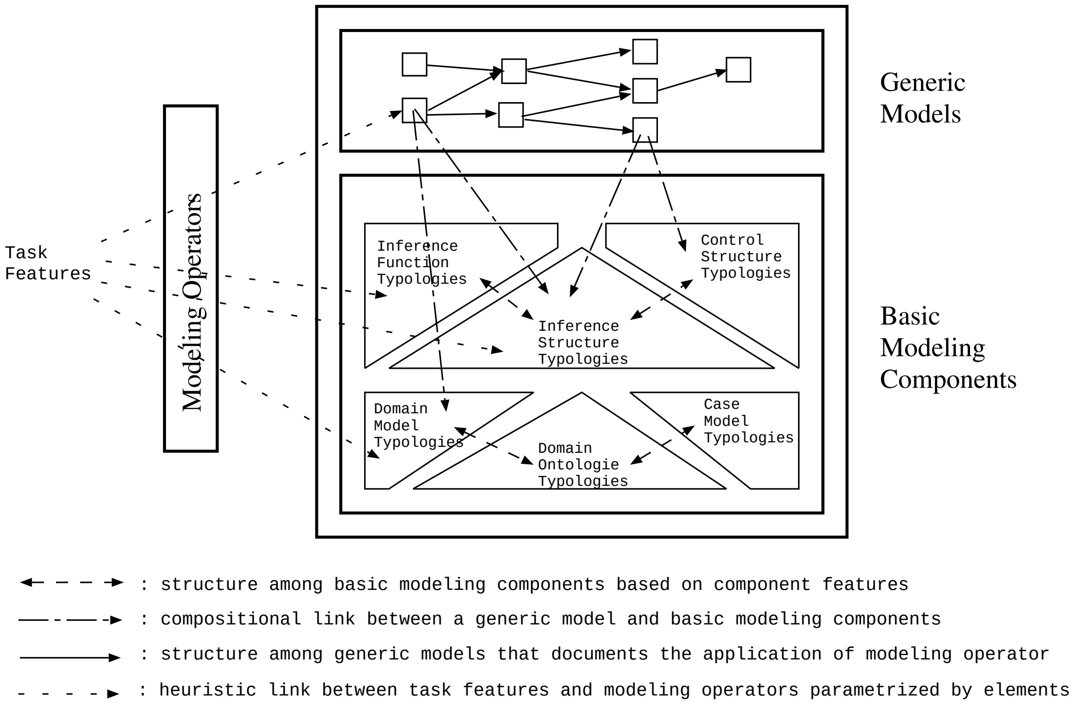



Figure 2.3: The library is structured in various ways to support its role of retrieving models and guiding the modeling process. The different structuring principles support access into the library (task features), organize the two categories of library elements (component features, typologies and modeling operators) and link them to each other (compositional links). (The layout of the library in the figure is not significant here.)

as the Model Selection Tool. The latter helps identifying appropriate library components on the basis of features of the task reality, while the CMED is the major tool for developing the expertise model.

Hence, the CommonKADS Library is not the only KADS-II tool used to build a model, and it cannot build it alone. The elements of the library are to be retrieved to be later used, completed and modified with an editor. Since the conceptual level is the central one, the editor which will be most important for the knowledge engineer developing an expertise model, is the Conceptual Model Editor - the CMED. The library need to play tightly together with this editor, since the role of the library is to supports the modeling process by providing knowledge items - or partial expertise model elements - organized and structured according to its intended use. The library plays an important role within the CommonKads modeling methodology, and it is important that the support tools is designed to facilitate and encourage an effective use of the library.

The mutual dependency and intended between the library and the support tools can be illustrated by making an analogy with an ordinary library. [6] If one needs to make a report, it is necessary to consult various literature sources: books, conference proceedings, journals, etc. To structure and fill one's document, it is possible to use pieces of text from articles or books - usually somewhat modified - and edit them into the document being prepared. The CommonKADS Library can be seen as the ordinary library in this analogy, and the Conceptual Model Editor can be compared to whatever tool used to prepare the document. The library is the support from which we may retrieve pre-prepared components and modify them in order to integrate them into the current version of our product (expertise model or document, respectively). The library and the

[6] The argument that follows applies also for formal level descriptions, with the Formal Model Editor.

tools to effectively utilize it help us to avoid "reinventing the wheel", and thus increase both the quality of the end product and the efficiency of the production process.

## 2.4 Summary

In this chapter we have described and motivated the basic design decisions for the CommonKADS model library. They can be summarized as follows:

- The library consists of two disjoint categories of elements: basic modeling components and generic models. The latter can be thought of as providing "form" (frames to be filled), the former as providing elements of "contents".
- Every library entity can be described in a verbal, conceptual and formal way. The conceptual descriptions are central, while the other descriptions - when available - can be accessed from it.
- The structure of the library of basic modeling components is derived from the necessary relationships between these components (type and structural). The library of generic models is structured according to different types of modeling operations over expertise models. These two means of structuring provide complementary guidance.

The remainder of this document focuses on specifying in more detail the basic modeling components, the generic models and the structure that relates these entities. The appendices provide examples of library contents.

# Chapter 3

# Practical Interpretation of the Common KADS Modeling Language

This chapter discusses the conceptual modeling language that is used in the library in two sections. Section 3.1 presents a summary of the CML as proposed in Deliverable I.1 [Wielinga et al., 1992a]. In order to realize the scheme proposed in Chapter 2 we must repartition the CML entities into a small set of basic modeling components. Section 3.2 shows how this is being done.

## 3.1 The CommonKADS Expertise Model

A CommonKADS expertise model consists of knowledge in three *knowledge categories*: *domain knowledge*, *inference knowledge* and *task knowledge*. Structure between these categories is realized by the so called *knowledge roles*. Knowledge about constructing a model within this framework is called *problem solving knowledge*. It consists of *problem solving methods* and *strategic knowledge*. The structure that prescribes the allowed form of any expertise model is also shown in Figure 3.1. An expertise model is described as a large componential structure (part-of) of knowledge items from predefined knowledge categories. For each category DI.1. specifies (potentially) a type-hierarchy for the legal representatives of this category, and prescribes the attributes that describe them. In the following subsections we briefly describe each of these constituents.

### 3.1.1 Domain Knowledge

Modeling the domain knowledge is concerned with representing the knowledge that is specific for the application that the problem solver reasons about. According to [Wielinga et al., 1992a] domain knowledge can be divided into *meta* domain knowledge and *application* domain knowledge. Application domain knowledge is knowledge about the specific system (object system) that is subject of the problem solving process. Meta domain knowledge describes the vocabulary and the structure of the application domain knowledge. [1]

The entities for modeling the application specific knowledge are:

- domain ontology

[1] Referring to meta knowledge as being domain knowledge is a bit unusual because this type of knowledge is independent from the application that is modeled and therefore *not* domain specific knowledge. The meta domain knowledge in fact specifies the knowledge representation that can be used for modeling the domain specific knowledge. We will not discuss this type of knowledge any further in this section.

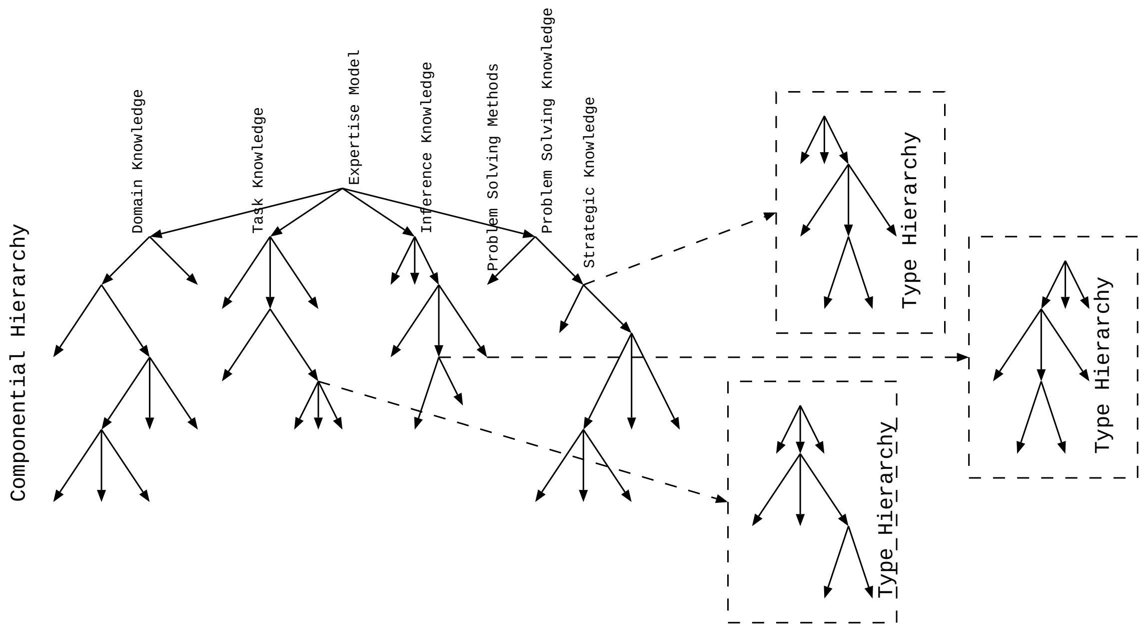



Figure 3.1: An expertise model is a large componential structure (part-of) of knowledge items from predefined knowledge categories. For each category DI.1. specifies (potentially) a type-hierarchy for the legal representatives of this category. The process of modelling can be viewed as filling this expertise model frame by pushing down the category representatives along the branches of their typologies until they are sufficiently specific.

- domain model
- case model

The domain ontology specifies the vocabulary that can be used for the construction of domain and case models. The domain model represents general knowledge of the domain, i.e. the knowledge that is independent from the specific system that is reasoned about. Some typical examples are models in terms of causal dependencies between symptoms and diseases in medical domains (causal model) or models that specify structural hierarchies for physical devices (structural models).

The case model represents the information about the specific system that has to be reasoned about. It presents the case specific data for the problem solving process. Four aspects can in this respect be distinguished: (1) initial data (provided before problem solving starts) (2) user input data (provided during the problem solving process) (3) intermediate information (data derived during the problem solving process) and (4) solution (required output of the problem solving process)

### 3.1.2 Inference Knowledge

By using domain knowledge (case and domain models) an inference is concerned with establishing a relation or a function between the various entities of the domain knowledge. The following aspects are important for specifying an inference (cf. [Wielinga et al., 1992a]):

- name
- dynamic knowledge roles
  - input role(s)

  - output role

- description (goal)
- static knowledge roles
- realization (or specification)

The name of the inference is merely a label which can be used as a unique identifier for the inference. The dynamic knowledge roles refer to the input and output of the inference, i.e. the domain entities between which the inference establishes a relation or a function (an inference usually has one output role, but may have multiple input roles). The description slot describes the relation between the input and output roles that is realized by the inference. The static knowledge roles refer to the domain knowledge that is used by the inference in order to establish that relation. Finally, the realization refers to a formal description of the inference, i.e. how the required relation between the dynamics knowledge roles is realized using the dynamic knowledge role. It is important to realize that an inference has no control. Both the activation itself as well as how often it must be activated is control knowledge that is imposed upon the inferences by certain tasks (see also task knowledge). In addition, control can not be exercised on the execution *within* the inference itself.

The following points specify additional constraints on defining an inference:

- An inference can be a 'relation' or a 'function' depending on the direction in which the dependencies between the dynamic knowledge roles may be established. In the case of a relation the inference can be used in both ways (non-directional), whereas in the case of a function the inference may only be used to derive the output form the input (directional).
- Whether or not an inference can be realized is determined by the ability to fill in the *realization* slot, in particular whether the required static knowledge roles are available. When this is not possible this means one is actually modeling a task that must be refined into smaller steps that can be realized by the available domain knowledge (see also task decomposition).
- An inference may introduce assumptions, i.e., characteristics of the domain knowledge that are assumed to be true (but not necessarily are). Assumptions are stated in terms of the inference specific terminology and they must be justified by the presence of certain task features of the application that is being modeled. For example, a single fault assumption can only be made if an application is not too critical (a task feature) and may be justified by the feature that time constraints apply.

The notion of an *inference structure* refers to a number of individual inferences being presented together. It presents an overview of the data-dependencies between the different inferences (in terms of knowledge roles) and the inferences that must be carried out. Crucial for an inference structure is that at least one of the knowledge roles (static or dynamic) of each inference is also a knowledge role of another inference.

### 3.1.3 Task Knowledge

According to [Wielinga et al., 1992a] task knowledge describes how a given goal can be achieved. It is essentially concerned with how inferences can be applied in order to realize problem solving goals. A task can in this respect be thought of as a 'shell' that encapsulates one or more inferences. It specifies:

1. which inferences must be carried out,
2. in what order inferences must be carried out, and

3. how often one inference or a sequence of inferences must be carried out.

Each task is characterized by a *task definition* and a *task body*. A task definition consists of:

- name
- case roles[2]
  - input role(s)
  - output role
- goal (description)

The name of a task is a label by which the task can be identified uniquely. The case roles refer to the input and outputs of the task. The goal (or description) refers to an intentional specification of the relation between the case roles that is realized by the task. The goal description therefore uses the terminology of the case roles.

A task body consists of:

- realization (by means of either:)
  1. an inference
  2. decomposition into subtasks (and/or subgoals ??)
  3. a transfer task
- control (either on:)
  1. on the inference (how often it has to be carried out)
  2. on the order of the subtasks
  3. when to present the task to the user

There are three ways in which a task can be realized. It may be the case that there is an inference that can make the required deductions in order to realize the goal of the task. If such an inference is not available then the task has to be decomposed into subtasks. Each of these subtasks has then to decide for itself if it can find an inference that can realize its goal. Finally, the task can 'become' a transfer task, i.e. the inputs of the task are presented to the user and the user is supposed to perform the task and present the required output to the problem solver again.

Most typical for a task is the control it imposes in order to realize its problem solving goal. Depending on the realization of the task three different aspects of control are important. When the task is realized by a single inference the control consists of determining how often this inference has to be executed. If the task is realized by decomposition into other task, the control specifies the order in which the subtasks have to be carried out. If the task becomes a transfer task then the control determines when the task has to be present to the user.

### 3.1.4 Mapping between Knowledge Categories

This section describes how the three knowledge categories as discussed above can be mapped onto each other.

[2]The difference between case roles and dynamic roles is subtle: dynamic roles are changed by the inference knowledge, whereas the case roles allow the task knowledge to "perceive" these changes in the case model.

- **Mapping between Inference and Domain Knowledge**
  The mapping between the inferences and the domain knowledge is realized by the dynamic and static knowledge roles. These knowledge roles define how a certain piece of domain knowledge is used by the inference. Firstly, these knowledge roles divide the domain knowledge into static and dynamic use. Secondly, the specific names each of these knowledge roles has (for example: symptom, disease and solution) further define the specific use of each piece of domain knowledge.

  Which pieces of domain knowledge map onto which static or dynamic knowledge roles is specified by the *model terms*. These model terms link a certain entity in the domain knowledge to a certain role of an inference. A typical connection between domain knowledge and inference roles is to map:

  - domain models onto static roles, and
  - case models onto dynamic roles.

  It seems also worthwhile to allow for domain models to perform dynamic roles, but it is impossible for an entity from a case model to perform a static role.

- **Mapping between Task and Domain Knowledge**
  The mapping between the tasks and the domain knowledge is realized by the case roles. These roles define which entities of domain knowledge are input and output for the task. Which pieces of domain knowledge map onto which case roles is specified by the *model terms*. These model terms link a certain entity in the domain knowledge to a certain case role of the task.[3]

- **Mapping between Inferences and Tasks**
  There is a relation between inferences and tasks. In the definitions given above a part of the task definition is similar to a part of the inference definition. In particular, the case roles (or dynamic roles) which specify the input and output roles are similar for both. However, the *contents* of the case roles (of a task) and the dynamic roles (of an inference) are only the same when the task is realized directly by the inference. Inferences and task differ with respect to the notions of static role, control and realization.

### 3.1.5 Problem Solving Knowledge

According to [Wielinga et al., 1992a] an expertise model consists of two major parts: a model of the application knowledge and a model of the problem solving knowledge. Domain, inference and task knowledge can be used to represent the application knowledge. Problem solving knowledge represents knowledge about the problem solving process. It is defined as a description of how to achieve a certain task, characterized by a goal and input/output specifications, through the generation of subtasks (goals) and/or the application of primitive inference actions. More specific: a problem solving method describes a way of decomposing a task in simpler subtasks and specifies control on these subtasks. A problem solving method consists of the following aspects:

- name
- goal
- (task) characterization
- control roles

[3] Only entities from the case model be inputs and outputs of the task. We can think of examples in which at least part of a domain model is required to specify the task definition. This would require a static knowledge role for tasks as well, a role which is currently subsumed by the case model. Of course, it is impossible that the output of a task feed into domain models (by definition!).

- subtasks
- control
- method definition
- task expression schema

In the case of a problem solving method that applies an 'primitive inference action' the control roles, subtasks and control are replaced by the 'inference relation used'. Unfortunately the 1.1 deliverable does not further specify the meaning of the different aspects that constitutes a problem solving method.

## 3.2 The CML versus Basic Modeling Components

In this section we present our repartition of CML entities into a basic orthogonal set of modeling components. Basically, we argue that similar functional units which are part of different entities from the CML should be singled out and represented in the library as independent units. These functional units then form the basis for the set of basic modeling components that are discribed in more detail in Chapter 4. This re-partition is motivated by the design issues and decisions for the library that have been discussed in Chapter 2:

- First of all we have been arguing that basic modeling components must not be too small so that they can be meaningfully indexed, and orthogonal so that redundancy is avoided. In terms of the componential structure this means that we do not consider as a legal library entity anything which is below a pre-defined (canonical) boundary.
- Second, the practice of modeling show that it is not always possible in advance to decide upon the role of an entity. A typical example is to choose between the status of "task" or "inference". While this may be clear in the final model, it is not initially. In terms of the type hierarchies of expertise model components this means that certain hierarchies have a common root (e.g., data flow element) that can serve as the temporary status of a knowledge item. As modeling proceeds the entity is specialized along the relevant branch until it finds its final place in the expertise model.

Figure 3.2 illustrates what is happening in terms of the original CML structure (compare with Figure 3.1).

### 3.2.1 Separating Data Flow, Decomposition and Control

We made the following choices for establishing the classes of basic modeling components to be stored in the library:

- *Separate out in the library the data-flow aspects from tasks and inferences.*
  Although terminology is somewhat different, the definition of an *inference* is similar to the definition of a *task definition* except for the notion of static roles and specification (realization). We therefore propose to create an independent modeling component that represents the common data-flow aspects of both inferences and tasks. Such a component represents the name of the task or inference, the input and output roles, and the goal or description.[4]
- *Separately represent decompositions from data flow elements*
  Multiple realizations of a data flow element (inference or task) can be thought of. It is therefore useful to separate in the library the the representation of a data flow element from the representations of the possible realizations (decompositions) of such an element.

[4] It seems a good idea to unify terminology for task and inference in DI.1 as well.

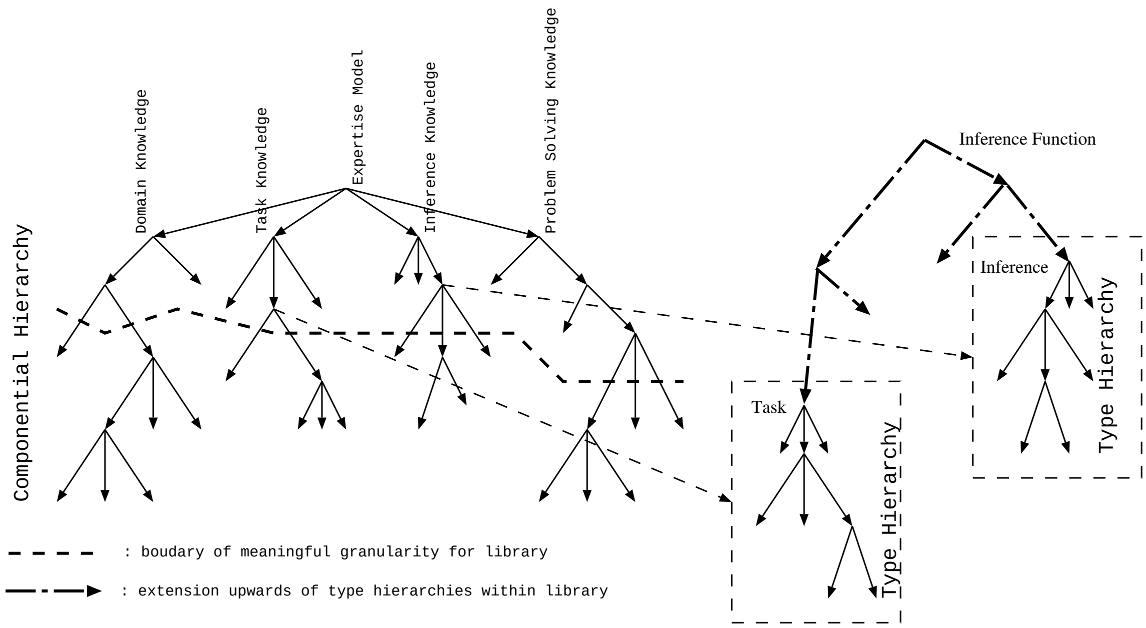



Figure 3.2: Basic modeling components define a fixed level of granularity for knowledge components to be meaningful library entities. Moreover, type hierarchies of different knowledge categories are merged by extending them upwards in order to account for the undetermined type of knowledge items in the library.

- *Separately represent control from (set of) data flow element(s).*
  Both a set of data flow elements as well as single date flow element can be controlled in multiple ways. It is therefore useful to represent in the library the possible control structures independent from the data flows to which they apply.

These choices lead to the following set of basic modeling components that deal with several (orthogonal) aspects of an expertise model:[5]

- *Inference functions* for data flow,[6]
- *Inference structures* for decomposition,
- *Control structures* for control,
- *Domain models, case models and domain ontologies* for domain knowledge,
- *Problem solving methods* for connection (or linkage) (see the following subsection).

### 3.2.2 Integration of Problem Solving Methods

In [Wielinga et al., 1992a] problem solving methods are not (yet) a fully integrated part of the CML. Their definition to a large extend overlaps with the definition that is given for representing

[5] One could consider that there is also redundancy in the domain modeling, between case and domain models. In order to avoid double modeling there one might introduce a unifying type for both domain and case model. In general, the library issues related to domain modeling are somewhat less worked out.

[6] This notion of inference function is, unfortunatly, not the same as the one used in DI.1 [Wielinga et al., 1992a]. For lack of a better name we use it consistently in this document. A similar remark applies to inference structures, which in the library play the role of both inference structure in the DI.1 sense and task decomposition.

task knowledge. More specifically, it is not clear how problem solving methods relate to the notions of inference structure and control structure.

For the library we will consider problem solving methods to be the dynamic counterparts of inference and task knowledge. Ideally, a problem solving method would specify:

- how a certain data flow element can be decomposed (or expanded) into a subset of data flow elements at a lower level of detail,
- which control structure is required on the data flow elements, and
- which requirements are imposed on the representation of the domain knowledge.

Not all three aspects must always be present in the definition of a method. Problem solving methods can be domain independent and thus not impose any requirements on the representation of the domain knowledge. Also we can think of problem solving methods that only provide us with an expansion or for that matter with a control structure.[7] Finally, problem solving methods may introduce recursive levels of decomposition, i.e. a configuration of data flow elements and control on these at more than one level of detail.

For the library the notion of problem solving methods may seem to be problematic, because they introduce a certain amount of redundancy with respect to sets of data flow elements and the control structures. The fundamental question is the following:

> *Can we think of problem solving methods that really (dynamically) generate sets of data flow elements and control structures* **or** *can we only describe problem solving methods in terms of their output (and consequently never capture the method itself)?*

For the time being we will adopt the latter[8]: a problem solving method can always be described in terms of data flow elements and/or control structures and/or domain knowledge requirements. We will use the notion of problem solving methods as being the entities that provide the *links* between the modeling components in the library. This choice follows the no-redundancy requirement as well as the integration requirement.

## 3.3 Summary

In this chapter we proposed a re-interpretation of the CommonKADS modeling framework and language for the purpose of its usage in the model library. This re-interpretation was motivated by the need to partition expertise models in a useful set of basic modeling components. The set of basic modeling components that we propose to store in the library is derived from three (representation) choices:

- Seperatly represent in the library the definition (data-flow) and realization or body parts of inferences and tasks;
- Separately represent in the library decompositions from data flow elements;
- Separately represent in the library control from (set of) data flow element(s).

Based on this, we have put forward a set of modeling components to deal with data flow, decomposition, control and domain knowledge. In addition, problem solving methods are being interpreted as the means to provide linkage between these different components. In this way we

[7] The so called weak methods (cf. [Rich, 1983]) are typical examples of problem solving methods that specify control structures.

[8] The exact interpretation of problem solving methods is a difficult issue to be solved by further Task I.1 work. Our minimal solution to the problem should not be considered as the final word on problem solving methods in CommonKADS.

have achieved a small set of basic modeling components that satisfy our needs. In the next chapter (Chapter 4) we will further elaborate on the detailed definitions for each of the basic modeling components in the library. The generic models that specify frames for putting together complete models from these components are treated in Chapter 5.

# Chapter 4

# Basic Modeling Components

In this chapter we describe in more detail the basic modeling components that were put forward in chapter 3: *inference functions* for date flow (Section 4.1), *inference structures* for decomposition (Section 4.2), *control structures* for control (Section 4.3), *domain models, case models and domain ontologies* for domain knowledge (Section 4.5) and *problem solving methods* for connection (Section 4.4) (see also figure 4.1). In the last section (Section 4.6) we discuss how the basic modeling components can be orgnanized in a series of *typologies*.

## 4.1 Inference Functions

Inference functions are Basic Modeling Components that represent data flow elements. The following aspects are important for defining an inference function:

- name
- dynamic roles
  - input role(s)
  - output role
- goal/description
- assumptions
- typologyreference

The name refers to a unique identifier of the inference function. The dynamic roles refer to the input and output roles of the inference function. The goal/description characterizes the input/output relation that the inference function realizes between the dynamic roles. Assumptions specify characteristics of the inference function with respect to its applicability. Typically they define requirements on the nature of the domain knowledge that the inference requires to work properly. What kind of static role is required by the inference function depends on the assumptions and the typology reference. [1] The typology reference refers to the typology on the basis of which the functionality of the inference is defined. [2] The ontology reference is optional but obviously inference functions that are not specified in terms of a certain ontology are less suited for supporting the knowledge acquisitions process.

[1] On the basis of the discussion in Section 2.2.3 one can argue that the typology reference is actually a component feature, expressing component necessity (in this case necessity of a domain ontology).

[2] Depending on the ontology in which the inference function is defined, these functions can be further distinguished into (1) basic (or problem related) inference functions and (2) canonical inference functions (see also Section 4.6). Inference functions that do not refer to a specific ontology will be referred to as library defined inference functions.

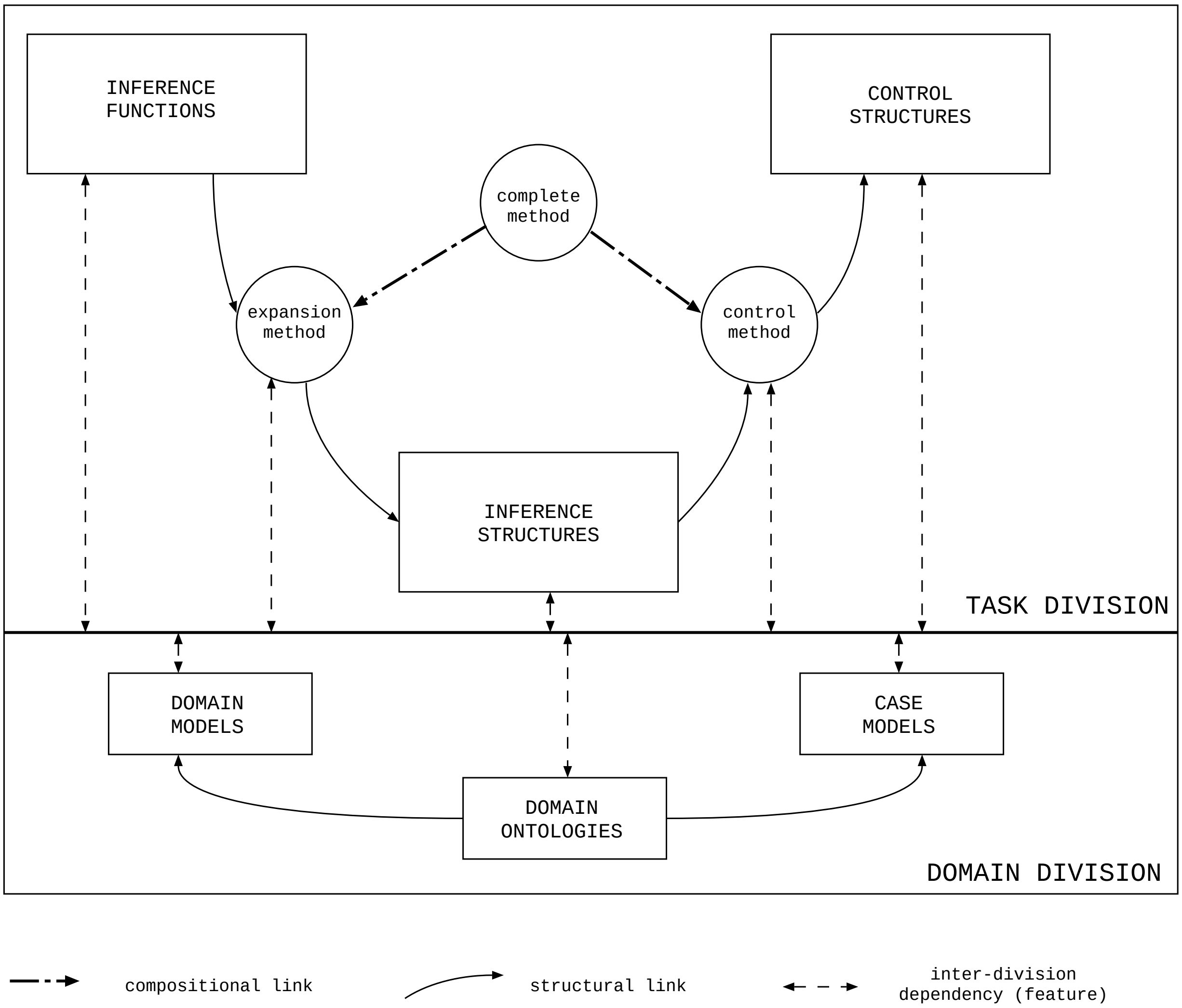



Figure 4.1: Structure of the library: basic modeling components

During the modeling process inference functions are transformed by the knowledge engineer into inferences or tasks (in the sense of [Wielinga et al., 1992a]) depending on whether *static roles* can be defined to realize the mapping between the input and the output roles. If the required domain model is not available the inference function will become a task that has to be decomposed into inference functions that can be realised by the available domain models.

## 4.2 Inference Structures

Decomposition is realised by an inference structure: a set of inference functions that share input and output roles. The relevant aspects of an inference structure are:

- name
- description
- a set of inference functions

- input/output dependencies between inference functions (each inference function from the set has at least one input/output dependency with another inference function from the set)
- assumptions

The name is again a unique identifier for the inference structure. The description refers to a statement in natural language about the characteristics of the inference structure. Each inference function from the set of inference functions that constitutes the inference structure is an inference function as defined in Section 4.1.

The inferences in an inference structure are grouped together in a connected input-ouput relation. This structure can be used to decompose an inference function at a certain level of detail into a set of inference functions at a lower level of detail. [3] The idea here is that the inference functions at the lower level of detail (possibly after a number of decompositions) will be detailed enough for realisation of the inference function by using the available domain models. How inference functions can be decomposed into an inference structure is defined by expansion methods (see Section 4.4).

The construction of inference structures may introduce new assumptions which did not apply to the individual inference functions of the structure. The assumptions slot can be used for this purpose.[4]

## 4.3 Control structures

Control structures are essentially a piece of control defined by a certain control language that includes things like 'while', 'if-then-else', etc. It imposes control on a single inference function or on an inference structure.[5]

Different control structures can be imposed on a single inference or inference structure, which is one reason why we choose to represent control structures as independent library elements. Although we can think of control statements independent of the specific set of inference functions to which they apply it is usually more appropriate to associate the control to a set of inference functions. In terms of library strcuture this simply means that the different control structures are linked by a component feature of component necessity to the inference or inference structure that they apply to. An additional and more fundamental reason why we represent control strcutures as seperate library entities is to accomodate the notion of a control method (see also Section 4.4). Control methods can be highly specific, i.e., pointing directly from an inference structure to a specific control structure. They can also be more general, however. For example the notion of data-driven control or hypothesis driven control are not specific to an inference structure but indicates a fairly specific type of control structures for a whole class of inference structures. These are therefore not control structures but specify ways - though maybe not algorithmicaly - of arriving at specific control structures.

## 4.4 Problem Solving Methods

The connections in the library are realized by the problem solving methods. At least three types of methods can be defined:[6]

[3] May the decomposed inference function be a part of the inference structure that realises the decomposition, i .e. be fully recursive?

[4] A criteria that could be imposed on the construction of inference structures is to require that only inference functions with the same ontology reference may used in a single inference structure (=canonical inference structures). This requirement would prohibit the construction of inference structures with inference functions at different (arbitrary) levels of detail. It would be premature, however, to enforce that restriction at this point.

[5] Control structures correspond to the notion of *task structures* in KADS-I.

[6] In order not to violate the orthogonality assumption, and based on the discussion in Section 2.2.1 complete methods are moved to the category of generic models.

- Expansion methods
- Control methods
- Role refinement methods

An expansion method associates an inference function at a certain level of detail to an inference structure at a lower level of detail and as such realises decompositions of inference functions. The relevant aspects of an expansion method are:

- name
- description
- inference function (to which the method applies)
- inference structure (the method delivers)
- mapping between input/output roles (of the inference function to input/output roles of inference functions in the inference structure)
- rational (of the method)

The name is again a unique identifier of the expansion method and the description is a statement using natural language about the method. The inputs and outputs of the expansion methods are described in terms of the inference function to which it applies and the inference structure that it delivers. The method must also specify how the input and output roles of the inference function that is decomposes can be mapped onto the input and output roles of the inference functions in the inference structure.

A method is not an arbitrary connection between an inference function and an inference structure, instead it is based on the features of each of these entities. In other words the expansion method is based on the characteristics of its input and its output. The rational is verbal description (or possibly a formal) of the reason 'behind' the method. It provides an account of why the expansion delivered by the method is an useful and/or interesting one.[7] Multiple expansion methods may of course be applicable to a single inference function.

A control method associates an inference function to a control structure.[8] The important aspects this method are:

- name
- description
- inference structure (to which the method applies)
- control structure (the method delivers)
- rational (of the method)

The definition of a control method is essentially similar to the definition of the expansion method. Although they differ of course with respect to their input and output.

Finally, there are methods that specify data refinement. These methods are a further specification of the mapping between input/output roles of one inference function to the input/output roles of another inference function. Usually the mapping is taken care of by expansion methods (see also definition of expansion methods).

[7] The 'description' and the 'rational' slot differ in the sense that the latter does incorporate an account of why the method is useful.

[8] A control method may also apply to a single inference function.

## 4.5 Domain Knowledge

Representation of the domain knowledge essentially follows the definitions discussed in the previous chapter (Section 3.1.1):

- Domain models
- Domain ontologies
- Case models

The connection between domain models and domain ontologies on the one hand and inference functions and inference structures on the other hand is realized by assumptions and the notion of typology reference. In particular, the latter specifies which domain models can be used for realising a certain inference function.

## 4.6 Typologies of Basic Modeling Components

In Chapter 2 we have introduced typologies as the means of structuring a class of basic modeling components. Typologies facilitate the construction of an expertise model from a generic model by indicating the refinements and alternatives that the library provides. Typologies can be based on various principles. For example the descriptions of the different basic modeling components that we have introduced in this chapter provides for a straightforward hierarchy based on a simple (frame-based) notion of specialization and inheritance. But other, more sophisticated typologies can be envisaged, for example based on ontological considerations. Table 4.1 lists two typologies on inference functions:

| Extension | Bibliographic References |
|---|---|
| Canonical Inference Functions | this report, [Breuker et al., 1987b] |
| Basic Inference Functions | this report, [Breuker et al., 1987b] |

Table 4.1: Two examples of typologies on basic modeling components.

**Canonical inference functions (typology):** The KADS-1 approach used a set of knowledge sources with a canonical character, that were in fact borrowed and adapted from KL-ONE. We can adapt the same typology to the KADS-II framework, generating the *canonical inference functions*. As the name says, they are a typology of inference functions that are defined by ontological distinctions. We discuss canonical inference functions in more detail in Appendix B.

**Basic inference functions (typology):** The same typology used to define the basic tasks in KADS-1 can be used to define a typology of inference functions, that we call correspondly the *basic inference functions*. Again, naming issues are open, but the names have to keep their correspondence to stress the common typology behind them. We may see the basic tasks as "filled" in basic inference functions, or redefine the basic tasks as a sum of a basic inference function, an inference structure and a control structure (i.e., a generic model). For the discussion on basic inference functions we refere the reader to the discussion on basic tasks in Appendix C.

Clearly other typologies can be derived along similar lines. For example CommonKADS abandoned the idea of using a KL-ONE ontology as the only possible basic ontology. Canonical inference functions as defined above are relative to the underlying representational ontology and a similar exercise can be done for any other ontological choice.

## 4.7 Summary

In this chapter we have described in more detail the basic modeling components and their relations (see also Figure 4.1):

- inference functions
- inference structures
- control structures
- domain models
- case models
- domain ontologies
- problem solving methods

In addition we have shown how typologies of such components allow for extending the library.

# Chapter 5

# Generic Models

As mentioned in chapter 2, the library consists of two basic types of entities: basic modeling components and generic models. The basic modeling components were discussed in Chapter 4. In this chapter we focus on the generic models. We will initially discuss what generic models look like (Section 5.1) and describe how this part of library is structured (Section 5.2). Section 5.3 expands on the relation between generic models and basic modeling components, and gives examples of classes of generic models that have been described in the literature. Several of the appendices in this document discuss such classes of generic models in more detail (role-limiting methods, problem solving methods for construction tasks, basic tasks), albeit not yet in a fully integrated way. The design of the generic model part of the library is illustrated in Figure 5.1.

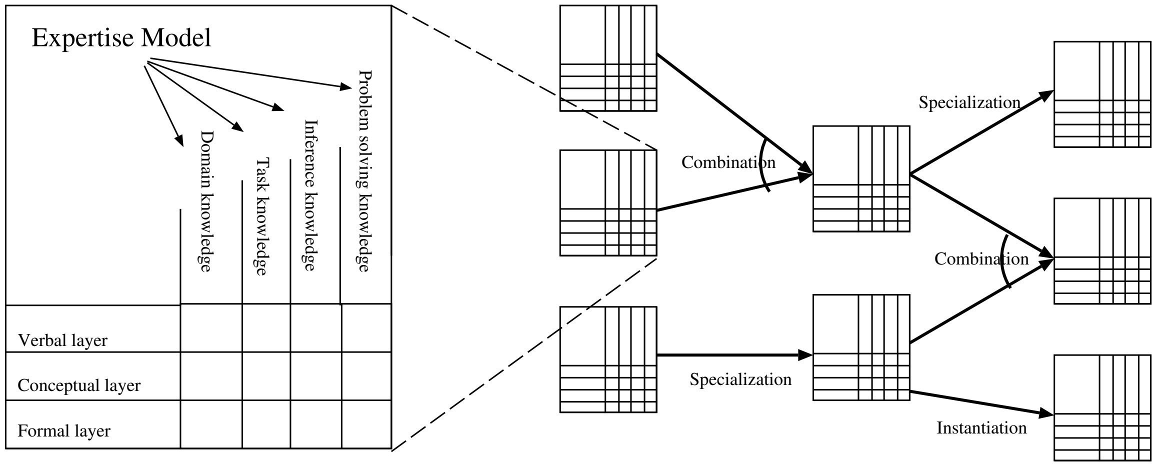



Figure 5.1: The library consists of partial (but in form complete) expertise models described at various layers of precision and organized in useful partial model development graphs.

## 5.1 The form of a generic model

The goal of modeling with the library is the construction of a complete and coherent expertise model. Generic models must serve templates and platforms for this process to start from. It is natural to think about generic models as partial (but in form complete) expertise models. The form of a generic model is therefore the following (directly borrowed from the decomposition of the expertise model as described in [Wielinga et al., 1992a; Figure 6]. The further structuring of the frame (a kind of "tree of knowledge") follows the specification of [Wielinga et al., 1992a] as discussed already in Chapter 3 and illustrated in Figure 3.1. Every generic model in the library is a partial instantiation of this frame. For some generic models only one slot may be filled in. For others all slots may be specified.

Basically there are two reasons for using only complete expertise model frames as generic models in the library. One is the relative interaction hypothesis [Wielinga et al., 1992a; Section 2], the other is flexibility in modeling [Wielinga et al., 1992a; Section 6]:

- The relative interaction hypothesis is one of the basic principles underlying the CommonKADS modeling approach. It has mainly implications for what counts as a generic model. The interaction hypothesis was introduced by Chandrasekaran to indicate that knowledge form and use could not be meaningfully separated. This hypothesis implies that the lowest meaningful level of granularity for generic models consists of a package of domain, inference and task knowledge. On the other hand the KADS-I view was that each of these three knowledge categories could be modeled relatively independently of each other. In CommonKADS the idea is that these are two extremes of a spectrum of possibilities and that the methodology must be able to accommodate any level of interaction between different knowledge categories. Consequently no implications follow for the level of granularity of a generic model. A model is generic if it is a useful platform to start the construction of a complete expertise model. Such a platform is now viewed as a partially instantiated expertise model frame.

- The KADS-I view is that the inference structure is preferred as "the initial or pivotal element in conceptual modeling"[1]. The Generic Task approach [?] would rather take task knowledge as the entry point. In Steels' Components of Expertise [Steels, 1990] domain knowledge plays a more prominent role in the initial phases of modeling, after which the focus may shift on task knowledge and decomposition knowledge. Clancey's NEOMYCIN [?] shows the importance of task structures while his notion of inference structure, as exemplified in the heuristic classification model [?], seems closer to taking the domain structure as a starting point. McDermott's analyses of problem solving [?] is centered around problem solving knowledge though it is closely coupled to task knowledge. But, no matter where one starts from it is necessary to fill in at least task, domain and problem solving knowledge (at least with inference and action methods).

It follows that it is not possible to make claims about the level of granularity of meaningful generic models, neither about the nature of the generic models that serve as entry points into the library. A single expertise model frame that can be instantiated to various levels of specificity accommodates these consequences. For example a KADS-I interpretation model becomes an expertise model with the inference structure and control structure filled in. In CommonKADS a variety of generic models co-exist [Wielinga et al., 1992a], for example role limiting methods, instantiated problem solving methods, generic tasks, and combinations of these. The advantage is that the methodology is not restricted to one "modeling paradigm" but accommodates a variety of modeling approaches. Providing for a certain type of generic model implies a bias toward a certain modeling practice, namely one in which the generic model is taken as the appropriate entrypoint in the modeling activity. Our approach avoids this bias.[2] It is true, of course, that coping with this variety requires more sophisticated modeling tools.

[1] Note by Joost Breuker, September 20, 1991

[2] There are other advantages of a totaly different nature. The fact that the library of generic models consists of expertise frames that match the full CML definition implies maximal expressiveness and transparency of library elements as well as an interface to modeling support tools that requires no additional translation. Incremental modeling and structural guidance (missing pieces are clear from the model) are additional advantages.

## 5.2 Modeling operations as library structure

What are the relations between the generic models? As argued in Section 2.1 an answer to this must be found by taking a pragmatic stance: how will the library be used? It was already pointed out above that the CommonKADS methodology is supposed to be more flexible with respect to the process of modeling itself. This does not imply that the contents of the library should be different depending on how you model. It does imply that the way in which the library is traversed and the model composed from partial ones is different.

The modeling process itself (as it is being studied in Task 1.5) can be viewed as a search process in which a model development graph is being build up. At each moment the CommonKADS expertise model frame reflects the status of knowledge level modeling. Modeling operators operate on expertise models. There are a number of such operations:

- specialization (e.g., specializing into a specific kind of diagnostic task)
- instantiation (e.g., filling in the contents of a domain model)
- combination (e.g., applying a role limiting method to a task)

This taxonomy can and should be refined, e.g. according to the type of specialization being done or on what knowledge category it works. The point is that the library must contain useful partial paths of the modeling process so that, once you get in, it guides the application of operators by offering different options and alternatives. In some cases, as illustrated in Figure 5.2, a complete model development graph can be obtained by a few initial selection steps (based on task features) and a final instantiation step (for example to fill in the domain model). In practice several model construction operations will need to be done outside of the library, of course.

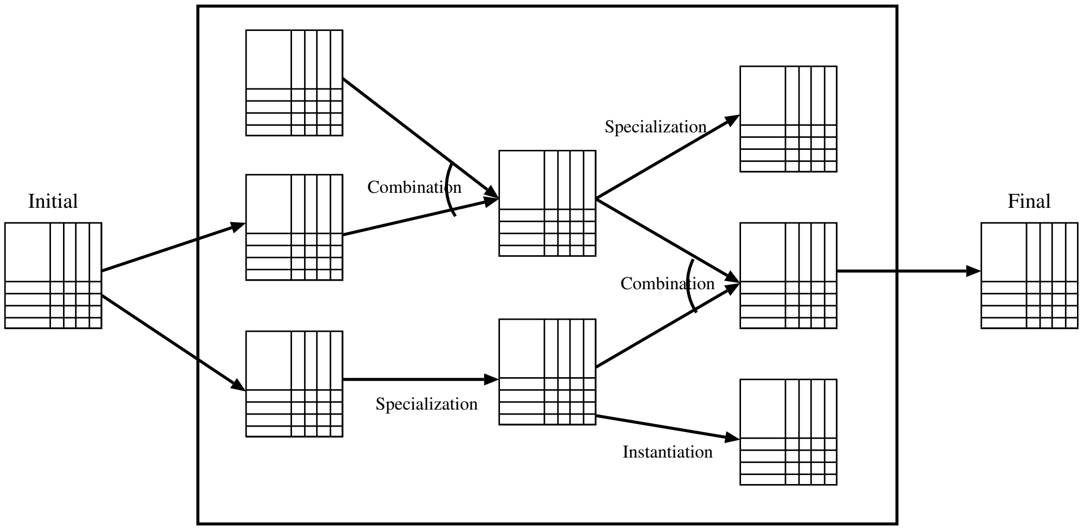



Figure 5.2: The structure of the library supports modeling by providing a set of useful partial model development graphs. These document the relation between generic models that can be obtained from one another by the application of modeling operators.

By making the modeling operations explicit we want to achieve a level of guarantee that all the necessary steps have been taken to ensure internal consistency of the model. A modeling operation is more than a single addition to a model. The intention to apply a modeling operation generates an "agenda" of tasks to be done to bring the model into a new internally consistent state. The application of a decomposition method (present in the problem solving knowledge) to a task (present in the task knowledge) will influence the domain, task and inference structures of the global model.

The structure among generic models thus allows for documenting the genesis of a generic model. For example, it could be documented that a model of systematic diagnosis is, for example, the result of applying a generate-and-test method to a hierarchical decomposition model. It is now for instance possible to trace back the origin of an inference structure as an "expanded" version of another one by the application of a decomposition method instead of an inference method. This gives us the effect of hierarchically structured inference structures for free. Also it is possible to move easily back and forth during the modeling process, for example having retrieved a fairly complete model of which only the domain organization is not quite right, it is possible to decompose it and recover the useful parts as the basis for a new combination step.

(Add arguments that this structuring is really more powerful than structure among basic components, e.g. for superimposing models).

## 5.3 Classes of Generic Models

It is interesting to notice that the two ways of extending the library (adding typologies or adding generic components) correspond to two ways of grouping components. On the one hand, generic models are made by defining, basically, meaningful sets of different basic components (e.g. one inference structure and one control structure). On the other hand, typologies of components are made by defining meaningful sets of the same component.

In the literature one finds several classes of generic models: interpretation models [Breuker et al., 1987b], role-limiting methods [McDermott, 1988], generic tasks [?] and so forth. Usually these generic models are tight to particular modeling approaches. It is tempting to introduce typologies of generic models, just as we did for basic modeling components. However, since generic models are made by basic ones, such typologies would probably refer (explicitly or not) to typologies of basic components. Therefore we do not define typologies of generic models since it would not add any power to the library (in fact, it could restrict it). We find it useful however, to refer to *classes* of generic models, since they can be related to notions from the literature (such as generic task). For guiding modeling however, the boundary of a class should not be "visible". At this point the knowledge engineer does not care what class of generic model he/she is working with.

Some classes of generic models are listed below.[3]

**Basic tasks** Although the KADS-1 interpretation models have been shown to have certain limitations, they have also shown to be very useful. The basic tasks here proposed are the correspondent to the interpretation models in the old KADS-1 library [Breuker et al., 1987b]. The name is used because they are based on exactly the same ontological (functional) distinction that generates the basic inference functions.[4] As generic model, they are a set of an inference function, an inference structure and a control structure. As a typology, not every set of these basic elements is allowed, but only some. The ontological distinction that generates the typology is system-centered. The detailed discussion of basic tasks is made in Appendix C.

**Role Limiting Methods** The discussion provided in [Duursma, 1992a; Duursma, 1992b] gives us a basis to include role limiting methods as proposed by McDermott [McDermott, 1988] in the library.[5] For what we could derive from these sources, role limiting methods are composed by two expansion methods and two control methods (that apply in succession), and *many*

[3] In the following description, many aspect relating to domain issues are not dealt with in detail. This should be completed later on, as soon as CML domain issues are better explained and resolved.

[4] The name may be somewhat misleading. Maybe *system-centered tasks* or *problem-related tasks* could be better; another option could be to maintain the old name (interpretation models).

[5] The study is particularly interesting because domain issues were addresses in detail.

features, both about task and domain. It is still open whether they are also a typology. Role limiting methods as generic models still have to be detailed and discussed, and the present definition is only initial. It is interesting to notice, however, that one differentiation factor between role limiting methods (other than the two step refinement in terms of our library) is their strong emphasis on domain issues. Since these are not treated in detail in this document yet, the full potential of role limiting methods is unfortunately hidden in our simple interpretation.

**Complete methods** A complete method is a combination of an expansion and a control method. It applies to an inference function and delivers both an inference structure and a control structure. Following the definition of expansion and control methods it is clear that a method must be characterized as follows:

- name
- description
- expansion method (specifying the inference function to which the complete method applies and the inference structure the method delivers)
- control method (specifying control structure the method delivers and the inference structure[6] to which the control applies)
- rational (of the method)

Complete methods may have an extended rational from their respective sub-methods (if it makes sense to make them explicit).

**Generic tasks** We may implement generic tasks in basically two ways. In the first, they would be complete tasks (i.e. a inference function + an inference structure + a control structure) at a certain (pragmatic) level of abstraction. In the second, they would also contain domain knowledge, or at least certain domain models or structures. If the first definition is assumed (which is less likely), we could regard basic tasks as defined above as particular types of generic tasks, defined with the help of a typology. Following the basic bibliographic reference in [?] the latter interpretation is more likely. Again, this interpretation has to be carried out in more detail, and an extensive discussion has to be included later on.

**Fields** The notion of field refers to a typology of application domains that can be regarded as the counterpart of the typology of basic tasks (inference functions) in the domain division. The concept of field is a new one[7]. It is still unclear what is their exact nature but as a first definition, we propose that fields are a set of ontologies and domain models. The latter is derived from the former. A full discussion of the issue is still lacking, and will be added in future refinements of this document.

Table 5.1 summarizes the examples of classes of generic models in the library.

The above examples indicate that we tacitly assume that any expertise model can be rewritten as a structure of basic modeling components. However, this does not mean that such a translation is easy or simple. Sometimes the whole theory these components are inserted in has to be reinterpreted in CommonKADS terms in order to allow such a translation. Given the proliferation of terms and theories in the field and the lack of unification between them, this is probably a very difficult task.[8] Therefore, a major effort in the re-interpretation of other modeling approaches and modeling components is necessary to enhance the CommonKADS library. The aim of this effort must be to document as much as possible the compositional links between a generic model and the basic modeling components that are at the basis of it.

[6] the inference structures of the expansion and control method should of course be the same.
[7] The notion of field has been proposed by Joost Breuker within KADS-II
[8] The problems of integrating Components of Expertise [Steels, 1991a] into KADS clearly illustrate this point.

| Extension | Definition | Bibliographic References |
|---|---|---|
| Basic Tasks | Inference function<br>Inference structure<br>Control structure | [Breuker et al., 1987b] |
| Role Limiting Methods | Inference functions<br>Expansion methods<br>Control methods | [McDermott, 1988] |
| Complete Methods | Expansion Methods<br>Control Method | this report |
| Generic Tasks | Inference functions<br>Expansion methods<br>Control methods | [?] |
| Fields | Ontologies<br>Domain Models | this report |

Table 5.1: Examples of classes of Generic Models in the CommonKADS Library

## 5.4 Summary

In this chapter we have elaborated on the second type of library entities: the generic models. We have argued that generic models are needed in order to accommodate the relative interaction hypothesis as well as for integrating different approaches to modeling. Of course generic models are not independent of basic modeling components. In fact the latter are often needed to further instantiate (fill) the former. In addition we have argued in favor of documenting the structural relations between a generic model and the basic modeling components that constitute its parts. In addition structure over generic models is provided by various modeling operators that represent in the library structure useful paths of modeling and document the origins of a generic model. Finally, we have shown how generic models can be used to capture different classes of generic models from the literature.

# Chapter 6

# Use of the Library

So far in this report, we have dealt with the structure of the library trying to maintain as much as possible a high degree of independence of issues related to their use. In this chapter, we will deal with some of these issues. However, we still try to maintain independence from approach specific aspects. That is, we will discuss some issues that relate the structure to the use, but the discussion will be done independently from the specific modeling method used. As much as we claim that with the proposed structure for the library it is possible to implement different modeling components, we claim that with the tools we provide here to specify how to use the library it is possible to use different modeling methods. This maintains a complete independence of the modeling approach – both structure and use, both modeling components and modeling methods.

Presently, the chapter is not completely structured, but has more a series of sections presenting some concepts or discussing some specific aspects. Also, some aspects here discussed are more of the scope of tasks I.1, I.5 and I.6, or involve of other aspects or subtasks within task 1.3.

## 6.1 Accesses and Routes

The library is structured to provide support for its use. This use consists of the retrieval of library components through navigation on the library items. We distinguish *routes* in this navigation. Some routes are structurally supported, and also indicated or prescribed to the user. Other ones are possible, but not recommended. The analogy with a real library would be to go directly to the shelves without searching before in the catalogues: it is allowed, but not recommended or supported. Within these two extremes – completely guided routes and completely unsupported ones – many alternatives may be used in the design of the library.

To support specific routes in the use of the library, we will distinguish structural *accesses* to the library. They are collections of components which are particularly useful as starting points to routes. We also distinguish between *primary* and *secondary* accesses. The primary ones are in some sense prescribed; i.e., recommended by the methodology. The secondary ones are accesses which are starting points to routes that are not recommended but still supported at some extent, and therefore may also be used. This distinction reflects a distinction (preference) between different routes (related at a certain extent to modeling methods).

## 6.2 Task Features and Component Features

In KADS-II, the basis for providing guidance and structure in the modeling process is the use of *features*. The details of their use is part of the task 1.5 work, but some aspects are relevant to be commented here.

In principle, to retrieve and use a component of the library means also to assume a certain set of constraints and premises that are intrinsically attached to the component. For instance, to use an expansion method 'generate and test' (and its associated inference structure), it has to be possible to test the hypotheses. This generic assumption is related to domain issues, and may need to be detailed in domain terms – i.e., it may be necessary to specify in more details which domain assumptions are consequence of this more generic one.

While retrieving a component in the library and adding it to a model being constructed, the user should have in mind that also some restrictions (features) have to be obeyed. The final conceptual model will consist of various conceptual knowledge items *and* a group of features. These features may of course be discarded at the end, since they are in principle only useful during the modeling process. However, taking into consideration the possible need for these features in later phases (e.g. the construction of the design model), it is more natural to keep track of them at least as a means of documentation.

Task features have been defined in Task I.5 as any characteristic of an application and its environment (the "task reality") that is relevant in selecting an appropriate model or model components for it. These task features have been grouped as features relating to the domain, the task purpose, the environment and to grounding. The set of basic modeling components that is being used in the library suggests another class of features, namely *component features*. These component features connect or link components of the library by relations of:

**(mutual) component necessity** When a library component can only be used when some other is also used.

**model necessity** When some component can only be used if certain features and/or elements are present in the conceptual model. The typical case is domain knowledge availability.

**(mutual) component exclusion** When the use of a library component excludes the use of another library components.

**model exclusion** When the use of a library component excludes the presence of other features and/or elements in the conceptual model.

See also figure 6.1.

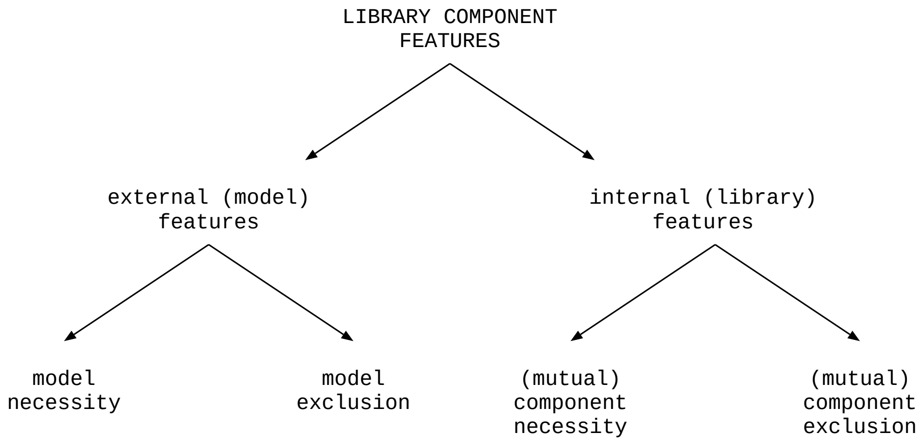



Figure 6.1: Typology of library component features

The *component necessity* and *component exclusion* features are particularly helpful in guiding the navigation through the library. For instance, the library should keep a trace of a consultation to verify mutual constraints and advise (positively or negatively) the retrieval of components using the features of the components already retrieved.[1] On the other hand, the *model necessity* and *model restriction* features are particularly adequate to help the user verifying whether or not a certain library component is compatible with the model being constructed. While the *mutual* features refer to constraints within the library, the *model* ones refer to restrictions external to the library – i.e., in the conceptual model being constructed.

Since some of the library components overlap in knowledge content, it seems that their features will too overlap to some extent. For instance, an expansion problem-solving method that points to a certain inference structure in the library will probably have many features in common with the inference structure itself.[2] More generally speaking, at any situation in which the choice of a component necessarily implies the choice of another (even if implicitly), the features of the second will be also in the first.

## 6.3 Operations on the Expertise Model

Since the use of the couple CMED/CKL concerns the process of building a conceptual model of expertise, it should be clear which *operations* can be done with respect to this model. These operations will reflect on different classes (types) of operations in the use of the couple CMED/CKL.

A first aspect to be specified is how (i.e., in which situation) will the user start to use the library. Since the construction of a conceptual model starts from the leaf tasks in the *task model*, we can assume that there is a task already defined (although only in general terms) when the model of expertise starts to be built. In other words, this task definition is the necessary connection between the task model and the model of expertise; the endpoint of the first and the start point of the second.

Therefore, the user will start the use of the CMED/CKL with an empty model of expertise with a single task. What should be filled is the task body of this main task (i.e., its inference structure and the control structure over it) and the domain knowledge necessary to operationalize the task (both filling domain knowledge roles and primitive inferences). These two needs correspond to the divisions in the library, and give us a basic distinction between two types of operations: the *task division operations* and the *domain division operations*. This distinction is not vacuous, but reflect an inherent different nature of the two divisions.[3] Note, for instance, that there is a mapping between domain and task/inference knowledge (provided by the domain roles), but there is no distinction between inference and task (at least in the library). Consequently, it seems appropriate to establish and use the two divisions proposed to the library also in the CML editor.

The inference and control structures are *layered*, i.e., they are constructed in many levels which provide details about how parts of the upper level will be accomplished. Therefore we can distinguish two other sub-types of operations in the task division: *single-level* and *multi-level* operations. Single-level operations work based on knowledge differentiation, i.e., they modify and adapt an existing element of the model[4], given some specific modeling needs. Multi-level operations are the ones which generate domain elements in a different level – by either decomposing (expanding) or assembling elements of one level into elements of another.

The types of operations on the conceptual model are summarized in figure 6.2.

## 6.4 How to Decide upon an Operation

In this section we will discuss the guidelines to the use of the CMED/CKL – in other words, how to decide upon an operation. As we have already mentioned, this guidance is primarily provided by *features*. Features are being used to guide in the application of operations on models. This

[1] It is a design decision whether or not to effectively forbid or oblige the user or only kindly advise.
[2] It is even debatable whether they can have different features at all.
[3] See, for instance, the discussion about knowledge roles in D1.1
[4] Which can be also components of the library, but not necessarily so.

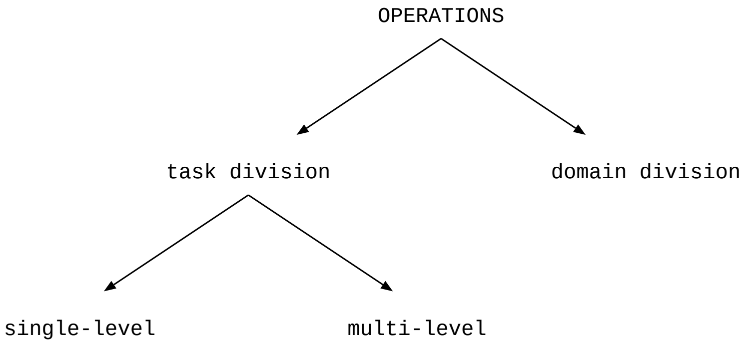



Figure 6.2: Types of operations on the library

guidance is provided by the so called *modeling guidelines* that have been proposed in Task I.5 work. The modeling guidelines can refer to either task or component features. It can also be conditional over the state of the model it is being applied to and the state that is being aimed at. When applicable, a modeling guideline gives a qualified recommendation for an operation that is itself parameterized over library elements. For example a modeling guideline may recommend to apply a method to a task, but it may also recommend not to apply it. The modeling guideline is completed with a way to verify the appropriateness of the operation (validation procedure). In summary a modeling guideline has the following attributes (see KADS-II/T1.5/WP/VUB/002/1.0 for a more complete treatment of modeling guidelines):

```
modeling guideline
   task-features                  <some task features>
   model-features                 <some model features>
   current-state                  <a model state>
   target-state                   <a model state>
   external-state                 <a milestone state>
   operator                       <a modeling operator>
   library-component              <some library-components>
   preference                     <a preference indicator>
   validated-by                   <a validation procedure>
   suggested-next-operator        <a modeling operator>
```

Operations make intrinsically different modifications in the set of features of the model. Modeling guidelines can help keeping track of the (hidden) implications of each operation. The typical example is the retrieval of a library component. When this operation is done, the features of the component being added are appended to the list of features of the model. By analogy, deleting an element implies that its related features have to be removed from the list. However, it will be difficult to keep track of this list of features when a modification operation is performed (particularly in the case of inference structures), for instance over a library component that was previously retrieved. Some of the features may refer to the overall element, and it is very difficult to know if they are still valid or necessary when the standard library component is modified, or if other features should come into place. The idea of connecting the features to the operations themselves could be used to solve this problem. For instance, when a canonical inference is deleted from a canonical inference structure, the library could, activated by the operation, verify which modification have to be done on the set of features of the working model because of the operation. At the same time, the kind of features that could change might be stored in the operation itself. A weaker version of this idea would be to store in the operations only the possible changes, in a way which indicates to the user which features might be affected without modifying them directly. This looks more coherent also, since the knowledge about which types of assumptions and restrictions are affected by an operation is more intrinsic to the operation itself than the specific change in the library elements.

Another kind of knowledge that can be used to decide upon an operation is a *modeling strategy*. This is the overall approach to knowledge modeling, reflected in the way the library is used. It is doubtful whether or not it is possible (or even within the scope of the project) to provide an intelligent overall guidance to the user in the form of a both active and passive intelligent help system (see, for instance, [Breuker, 1990], it seems reasonable to embed these possible strategies in the library in some way. They would probably be represented as *plans*.

In summary, the information about how to decide upon an operation would be stored as:

- task features
- component features
- model state
- modeling strategies and plans

## 6.5 Entering the Library: Basic Tasks and Fields

The major difference between CommonKADS Library of Re–Usable Models and its predecessor – the KADS–I Library of Interpretation Models[Breuker et al., 1987b] is that it contains also domain models, besides models of the reasoning strategies, i.e. the CommonKADS Library of Re–Usable Models has a task division which contains the reasoning models, and a domain division filled with basic models of domain knowledge (see Chapter 3).

The access to the models in the KADS–I Library of Interpretation Models was provided by a taxonomy of Generic Tasks. This taxonomy was based upon an 'ontology' derived from Clancey's view that a KBS is a 'model of some system in the world' [Clancey, 1985; Clancey, 1992]. From this notion the KADS–I project derived a pseudo–'ontology' in which the Generic Tasks reflected the notion of '*what can happen to a system model*', or, what functions (operators, procedures) can be applied to a system model. In the CommonKADS Library of Re–Usable Models we will maintain the same 'ontology', but we will extend it now to domain models as well. For the domain division the taxonomies are simply based upon the question *what types of 'systems in the world' (system models) can be distinguished.*

### 6.5.1 Task Division routes

Both questions have further implications. These implications create the refinement **routes** in the CommonKADS Library. As stated in Chapter 3, *inference functions* as basic components of the description of reasoning strategies can be refined by *expanding* them by applying expansion (p.s.) methods. Expansions answer the question what we mean by some function, by intentional definitions, i.e. one can say that a function or task is redefined, rather than decomposed.[5] The inference structures that result from expanded inference functions can be *elaborated* by applying control (p.s.) methods, which add control to the structures, thus resulting in task structures. The expansions and elaborations create the stepwise refinement routes in the Task Division of the CommonKADS Library. In general, access and refinement are accomplished by checking whether specific features

[5] This looks as a neat distinction, but the consequences look strongly the same. (Re–) definitions consist of other terms. The notion of 'other' means that if recursion or cyclic definitions are forbidden, a definition is a composition, and the 'definiens' (i.e. the thing to be defined) is decomposed in other terms. From the point of view of looking back from the result, decomposition and redefinition is the same; from the point of view of dynamic problem solving the two are not the same, because it makes more sense to talk about 'redefining a problem in new terms' than to say that one 'decomposes a problem', because by itself a problem has no parts other than what is given and what is asked.

are present in the domain of analysis. These features can be part of the model –e.g. in diagnosis whether the solution is a malfunction or a defective component–, or can be assumptions that are implied by selecting a model –e.g. the presence or constructibility of a taxonomy of malfunctions. When these assumptions refer to components in the other division of the CommonKADS Library the (p.s.) method is called a *role–limiting method* [Marcus, 1988]. Because the grain–size level of the CommonKADS Library is not fixed[6] the routes may re–enter. An inference function may be expanded by a basic task. For instance, the specification of expected values (norms) in diagnosis may be expanded to 'prediction of behavior', which is a basic task. For that reason it seems also more adequate to use the term expansion rather than decomposition for the refinement steps.

Besides stepwise refinement routes, there are also packaged refinement routes. As tour–operators may offer all–included tours, some p.s. methods –in particular the role–limiting methods– provide in one go a more or less complete model of expertise for some (sub)task. These compound p.s. methods stand for what is traditionally understood by a problem solving method.

The task–division of the CommonKADS Library terminates in *canonical inference functions*. These make up the regular interface to the domain division, because the (re–)definition of of these functions in terms of meta–domain concepts, or to be more precise, in terms of the most abstract ontologies of domains. The most abstract domain ontologies are vocabularies for knowledge representation. In other words, the intentional semantics of the inference functions ends by referring to entities that belong to the most abstract ones in the Domain–Division.

An example of canonical inference functions and their derivation from knowledge representation terms, is the KADS–I Library of Interpretation Models typology of knowledge sources [Breuker et al., 1987b]. In the CommonKADS Library different typologies may be used. The one developed in the KADS–1 project is revised and presented in Appendix B. It is the current default, but new ones may be created in the course of further 1.3. activities.

### 6.5.2 Domain Division routes

The components of the Task–Division consist of inference and task–structures. Those of the Domain Division are (pre–) ontologies. An ontology is the distinctive vocabulary of a domain. Because terms in the ontologies can be relations -besides the typical objects, an ontology may be structured, and stand for abstract 'tops' of domain structures.

For the Domain–Division the refinement routes are less evident. Fields, which are the access to the domain division, are selected on the basis of their type (features). A field reflects one or more domain theories as discussed in Deliverable 1.1. [Wielinga et al., 1992a]. Domain models form the next refinement step. A domain model is a functional view on a domain theory [Steels, 1990]. The ontologies of domain models may be subsets of those of domain theories (fields), but at the moment we have not enough evidence to say that this is the case in general (see below on problems with defining domain models).

In the Domain–Division of the CommonKADS Library the refinement route can be stated as follows:

field → domain model → (canonical) inference function (support role)

In the Task Division of the CommonKADS Library, components are defined in terms of other, more detailed components which are finally grounded in ontologies via canonical inference functions. In the context of Deliverable 1.1. this ontology would be called 'basic model structure' (or: '(domain) model (structure) schema') (see figure 7 [Wielinga et al., 1992a]), which is more or less of what is meant by the basic, conceptual vocabulary of a knowledge representation formalism (see further). Ontologies are part of the Domain Division, therefore, the canonical inference functions are the natural interface between both divisions. In models of expertise the linking between the inference layer and the domain layer consists of the *domain mappings* of roles and functions. In

[6]Except for primitive inference functions.

the domain–independent definitions of canonical inference functions the mapping refers to highly abstract knowledge representation formalisms. In terms of Deliverable 1.1. (version 3), these domain–independent ontologies are 'meta–domain knowledge'. In this way, there is a grounding of the Task–Division of the CommonKADS Library. Conceptual entities can be grounded in unambiguous semantics in two ways: by reference to *computational* realizations (e.g. [Woods, 1975]), or by translation and validation into (logical) formalisms. Research in AI on knowledge representation is concerned with both routes. Therefore, we recommend that typologies of canonical inference structures are to be based on validated and well understood knowledge representation ontologies. Note also that the recasting of the components of the CommonKADS Library, expressed in CML, into FML (Formal Modeling Language) in the next stage of the 1.3. Task, has also this validation role.

In principle, the knowledge representation ontology to be used for grounding the CommonKADS Library should be the meta–domain knowledge as presented in Deliverable 1.1., Sec. 6.2, i.e. the CML for the domain layer. As far as this one is established (see Sect. A.1. of D 1.1. (version 3), it overlaps with the one presented in Appendix B.

Ontologies for domain models will be expressed in CML domain layer terms. However, for domain models and fields also other means may be available. The ontology of a domain theory may have also a *calculus*, which combines a formal and computational grounding. For instance, a number of calculus for time, space, causality, mechanics, etc. may be found in the literature. This is not only the most valid grounding, but it also supports the knowledge engineer, because he may get the description of a (specialized) inference engine for free. Wherever possible, the library should contain references to these calculus. It should be noted that the grounding of the ontologies only in terms of the domain CML is a weak and minimal one. A calculus is the most valid foundation one can have for the ontologies in the library.[7]

## 6.6 How to Specify a Modeling Approach

To understand how to implement or specify a modeling approach or a modeling method, we should first make clear what they are – more specifically, what such things contain and define. Of course,

[7] There are some problems with the notion of domain models that need to be resolved. Domain models are functional partitionings of domain theories. Steels presents as examples: functional, causal, teleological, structural, topological, etc. [Steels, 1990]. This functional perspective poses problems in the CommonKADS Library, because the functional viewpoint is in fact part of the Task–Division. If we want to have only 'interaction' in Chandrasekaran's sense between domain knowledge and the functional perspective at the level of canonical inferences we should avoid these functional perspectives in the Domain–Division. One may argue that this is not much of a problem, because one may use straightforward function–dependent links between the two divisions: a kind of shortcuts, where some inference function directly pre–supposes some domain model, as e.g. in role–limiting methods. However, the real problem is then that a model with the same name and role (function) may have completely different ontologies (definitions). For instance, a causal model in medical diagnosis has or may have a completely different ontology, than a causal model for troubleshooting audio–equipment. Moreover, the same ontology may be used for different functions. A causal model and a teleological model may have the same ontology –in an application: the same knowledge base–, but may only differ in the reasoning perspective. For instance, it rains to make the plants grow vs. plants grow because it rains. Because generic (re–usable) domain models may neither have a unique ontology, nor an exclusive one, domain models may not be the right components for the CommonKADS Library. This does not mean that the notion of domain models may not be very useful in constructing models of expertise. On the contrary, and this notion should be used in partitioning application domain knowledge, but one may doubt whether they can exist as 'meta–domain knowledge' elements in the CommonKADS Library. In the 1.1. Deliverable, Sec.6.2, domain models are part of the application domain knowledge. However, the ontology of models is assumed to be part of the meta–domain knowledge. At least in Steels' view such generic role of domain models is not intended [Steels, 1990], p. 39–40.

What domain models of the same function *may* have in common are relation terms. For instance, causal models should have some causal relationship explicitly represented, or derivable. Causal relations come in large varieties, but their meaning is at least a non–domain dependent topic of research in AI. Therefore, we may propose to replace the notion of domain model as a refinement step in the Domain–Division by the notion of *relation*. In KADS–I a similar view was held but it went never beyond subsumption and aggregation ('part–of') relations. See for more and better details [Bredeweg, 1992], p. 45–46.

this is a matter of hard and hot discussion in the AI community (and particularly in the Knowledge Representation and Acquisition communities). Therefore, we will only show our point of view over the subject: we do not want to enter that debate, but only use some concepts to aid our (local) task.

In the first place, it is important to make clear that we are dealing with *knowledge* modeling approaches and methods. All approaches we are analysing have a common goal, which is to produce knowledge-based systems through the construction of a certain number of knowledge level models.[8]

A knowledge modeling approach can define a number of things:

- A *knowledge theory*;
- An *ontological view* on knowledge based on this theory;
- A *knowledge representation scheme* based on this pair theory/ontology;
- A number of *knowledge level models*;
- A number of *knowledge modeling components*;
- One or more prescribed ways to use these components in order to build the desired models – i.e., a number of *modeling methods*;

Only some of these concern us, since the library only involve or support some of them. The two last items are the ones we should pay attention. In the first place, the knowledge modeling components are developed on top of a knowledge representation scheme (sometimes a language), and they represent a partition on the knowledge to be modeled into a number of items. This partition does not represent the same ontological distinctions made by the representation scheme itself, neither (commonly) have a strong theoretical status. they are a more pragmatic partition, conceived to help the modeling process. Using the terminology proposed by Lenat and Guha [Lenat and Guha, 1990], the knowledge representation scheme is epistemologically adequate, while the knowledge modeling components reflect a highly pragmatic preoccupation. That is, the first is designed to be as much as possible *independent* of the modeling process itself (i.e. its use), while the second has the opposite design concern. Of course, between these two extremes many levels can exist, with various schemes or sets of components at different levels of epistemological and pragmatic adequacy.

In summary, the modeling components are, within the context of a modeling approach, the tool which is designed to be adequate for the pragmatic concerns – i.e., their efficient and practical use in the modeling process. The use itself has also to be designed, and this is done with the *modeling methods*. They give a more thorough specification of how the modeling process may (or should) be done. In particular, they specify how to use the modeling components to build the desired knowledge-level model. That is the relation between the modeling components and the modeling methods within the context of a knowledge modeling approach.

For what refers to *a* library only (not necessarily *our* library), to specify or implement an approach it is necessary to specify the base knowledge representation scheme, the set of modeling components and the modeling methods. Since our library imposes a knowledge representation scheme (the pair CML/FML), the first step is fixed, and only the latter two remain. Thus, to specify a knowledge modeling approach in the library, it is necessary to:

- Specify the set of modeling components used in the approach;
- Specify the modeling methods used in the approach;
- Interpret the modeling components in terms of Common KADS CML, and more specifically in terms of the library basic components;

[8] Knowledge level as proposed by Newell [?].

- Use this interpretation to adapt (if necessary) or simply re-specify the modeling methods to Common KADS terms.

One interesting open question is the lack of a language or framework for the specification of modeling methods. Until now, although some of them have been formally treated or presented, the AI community is far from a consensus of how to describe a modeling method. The work presented here suggest some results in this direction, but a full blown study is certainly out of our scope. Evidence is growing that the effective usage of generic models is far from being solved. In [?] essential problems with selecting and using generic components are pointed out, based on practical experience:

1. The discrimination problem.
   Several generic models may apply. How to select the most appropriate one?

2. The overgenerality problem.
   Selected model is too general to give useful support. How to arrive at a more specialized model?

3. The modification problem.
   Selected model is not directly applicable. How to modify it?

4. The terminology problem.
   Selected model is described in a strange vocabulary. How to map it onto the ontology of the application?

5. The risk problem.
   Selected model may turn out to be wrong. How to minimize the costs of selecting a wrong model?

Based on these problems a set of guidelines for a knowledge modeling methodology is being suggested [?]:

- Combine bottom-up and top-down analysis.
  Bottom-up analysis may provide useful pointers to, and initial assessment of, generic models.

- Multiple selection criteria are needed.
  In addition to knowledge availability (the initial criterion used), other criteria suggested are: domain complexity, the expert's representation of the domain, information reliability, real world interaction, domain stability.

- Provide composable models.
  Task features should also point to schemas of combinations for generic models.

- Integrate modeling and model validation.
  Check expectations of real world behavior indicated by selected models.

- Provide for an extensible library, with annotations based on experience.
  Experience from using the library must be captured.

- Explicitly document the assumptions of generic models.
  Underlying assumptions should be used in the model selection process.

- Allow for partial models as end product.
  Weaken the role of the knowledge level model as a complete intentional specification.

The library as it is being proposed provides for the facilities to deal with these issues.

## 6.7 Summary

In this chapter we have discussed a number of issues related to the effective usage of the library. We have elaborated on the concepts of task feature, model feature and modeling guideline. We have also put forward some ideas on how to specify modeling approaches and have pointed out some points of current concern in current work on using generic models as a basis for a development methodology.

# Chapter 7

# Conclusion and Future Work

In this document we have described the present state of the CommonKADS model library. We have described the entities of the library, the structure of the library and we have described how it should be used. The appendices describe several library entities in considerable detail, and illustrate how practical modeling can feed back into the library.

The framework on which the library is based takes the ideas presented in D1.1 [Wielinga et al., 1992a] one step further by integrating them in order to facilitate alternative approaches to modelling problem solving expertise. The highlights of the library design are the following:

1. The library consists of two disjoint categories of elements: basic modeling components and generic models. The latter can be thought of as providing "form" (frames to be filled), the former as providing elements of "contents".

2. Every library entity can be described in a verbal, conceptual and formal way. The conceptual descriptions are central, while the other descriptions - when available - can be accessed from it. In this document we only focussed on the conceptual descriptions.

3. The basic modeling components are one partition of the expertise model in elements small enough to provide the desired flexibility by their recombination. Moreover their contents do not overlap and they are not too small to allow for meaningful indices to them.

4. Among the basic modeling components it is distinguished between components storing knowledge about data-flow (inference functions), typical patterns of data-flow (inference structures), control (control structures) and domain models. In addition, the problem solving methods specify how these components are related.

5. The *generic models* are frames that represent a class of complete expertise models. They capture useful combinations of basic modeling components and serve as templates for the construction of complete expertise models. This notion of generic model provides for any kind of model to support a modeling approach (e.g., generic tasks, role limiting methods, interpretation models, complex problem solving methods, and so forth).

6. The structure of the library of basic modeling components is derived from the necessary relationships between these components. The library of generic models is structured according to different types of modeling operations over expertise models. These two means of structuring provide complementary modeling suport, guided by task features.

The framework can be used to facilitate several alternative approaches to supporting the construction of knowledge based systems. In particular, the KADS-II project is supposed to establish a further elaboration of the, sometimes preliminary, ideas development in the KADS I project

(P1098). It is therefore interesting to see how the framework, underlying the structure and contents of the library, deviates from the KADS I approach [Wielinga et al., 1991]:

- The grainsize level at which an inference structure is defined is not fixed, but may depend on the characteristics of the domain knowledge that is used (relative interaction hypothesis).
- Decomposition of data-flow dependencies. Instead of consisting of a single inference structure (and a control structure) the conceptual model of expertise may constitute of a number such inference structures at different levels of detail, with each of these structures having their own control.
- The notion of problem solving methods, which is well known and often used in the AI community is now an integrated part of the modeling framework. The role of problem solving methods was not well understood in the KADS 1 framework.

As discussed in section 4 the notion of inference structures was one of the corner stones of the KADS 1 methodology. The framework that we have presented for constructing the KADS II library still allows this approach, i.e. a knowledge engineer may still build models of expertise by adopting this point of view. However, the framework also allows the construction of expertise models based on different views, as for example, step-by-step task decomposition (cf. [Steels, 1990]).

The work on the library is far from being finished. As has been argued in the introduction, the final library will be the result of a collective effort of the entire KBS community. Nevertheless we would argue that the beginnings have been made for a comprehensive integration of all expertise on modelling in the KBS community. To further shape and fill the library the following will be done:

- further work within KADS-II (toward the second version of this deliverable),
- import from other projects (Vital, Construct, Quick,...),
- import from users (e.g., KADS user group meetings).

Despite this loaded agenda for "future work", we believe that we have made considerable progress toward the goal of a real integration of modeling frameworks and approaches. The proposed library design is, we think, not a compromise but an elegant and flexible solution to a variety of difficult problems. We therefore hope that the contents of this Task I.3 report will feed back into the final version of DI.1.

# Appendix A

# BNF of the CommonKADS Library

**Bert Bredeweg**
**Joost Breuker**
**Andre Valente**

UvA

*Version july 1, 1992, André Valente*

In this appendix, we present a BNF of the library. It is divided in two parts: in the first we present a BNF of the basic modeling components as defined in chapter ??, and in the second we give teh BNF of the generic models as defined in chapter ??. These chapters are supposed to define the semantics for the language here described, with the help of the definition of the CML in the D1.1 report [Wielinga et al., 1992b].

The basic source of this BNF is the latest available (internal) version of the CML. The latest externally available version of the CML is in [Wielinga et al., 1992b]. The BNF presented here is essentially a modification of the one presented there.

It should be noticed that the descriptions herewith assumes that some sort of language will be developed to describe features, and that there is a BNF for formal knowledge in which we do not enter into details, define dwithin task 1.2. Therefore, the description of the library elements is given essentially with its conceptual and verbal content, and with the features described in a non-structured way (i.e., text). As the work in task 1.5 develops into a structured language, we should complete the text here defined, or refer to 1.5 deliverables.

## A.1 Basic Library Components

In this section, we describe the syntatic structure of the basic modeling components as described in chapter ??. We describe here, in detail, only their conceptual definitions. We assume that the BNFs for *feature* and *formal-knowledge* are provided by tasks 1.5 and 1.2, respectively.

### A.1.1 Some initial definitions

verbal-definition ::= **verbal-definition** text. (1)
formal-definition ::= **formal-definition** formal-knowledge. (2)
component-features ::= **component-features** feature⋆. (3)
name ::= **name** text. (4)
description ::= **description** text. (5)

### A.1.2 Four-part structure of the basic components

inference-function ::= **inference-function** (6)
conceptual-inference-function (7)
verbal-inference-function (8)
formal-inference-function (9)
inference-function-features. (10)
verbal-inference-function ::= verbal-definition. (11)
formal-inference-function ::= formal-definition. (12)
inference-function-features ::= component-features. (13)

control-structure ::= **control-structure** (14)
conceptual-control-structure (15)
verbal-control-structure (16)
formal-control-structure (17)
control-structure-features. (18)
verbal-control-structure ::= verbal-definition. (19)
formal-control-structure ::= formal-definition. (20)
control-structure-features ::= component-features. (21)

expansion-method ::= **expansion-method** (22)
conceptual-expansion-method (23)
verbal-expansion-method (24)
formal-expansion-method (25)
expansion-method-features. (26)
verbal-expansion-method ::= verbal-definition. (27)
formal-expansion-method ::= formal-definition. (28)
expansion-method-features ::= component-features. (29)

control-method ::= **control-method** (30)
conceptual-control-method (31)
verbal-control-method (32)
formal-control-method (33)
control-method-features. (34)
verbal-control-method ::= verbal-definition. (35)
formal-control-method ::= formal-definition. (36)
control-method-features ::= component-features. (37)

| | | | |
|---|---|---|---|
| refinement-method | ::= | **<u>refinement-method</u>** | (38) |
| | | conceptual-refinement-method | (39) |
| | | verbal-refinement-method | (40) |
| | | formal-refinement-method | (41) |
| | | refinement-method-features. | (42) |
| verbal-refinement-method | ::= | verbal-definition. | (43) |
| formal-refinement-method | ::= | formal-definition. | (44) |
| refinement-method-features | ::= | component-features. | (45) |

| | | | |
|---|---|---|---|
| inference-method | ::= | **<u>inference-method</u>** | (46) |
| | | conceptual-inference-method | (47) |
| | | verbal-inference-method | (48) |
| | | formal-inference-method | (49) |
| | | inference-method-features. | (50) |
| verbal-inference-method | ::= | verbal-definition. | (51) |
| formal-inference-method | ::= | formal-definition. | (52) |
| inference-method-features | ::= | component-features. | (53) |

| | | | |
|---|---|---|---|
| domain-model | ::= | **<u>domain-model</u>** | (54) |
| | | conceptual-domain-model | (55) |
| | | verbal-domain-model | (56) |
| | | formal-domain-model | (57) |
| | | domain-model-features. | (58) |
| verbal-domain-model | ::= | verbal-definition. | (59) |
| formal-domain-model | ::= | formal-definition. | (60) |
| domain-model-features | ::= | component-features. | (61) |

| | | | |
|---|---|---|---|
| domain-ontology | ::= | **<u>domain-ontology</u>** | (62) |
| | | conceptual-domain-ontology | (63) |
| | | verbal-domain-ontology | (64) |
| | | formal-domain-ontology | (65) |
| | | domain-ontology-features. | (66) |
| verbal-domain-ontology | ::= | verbal-definition. | (67) |
| formal-domain-ontology | ::= | formal-definition. | (68) |
| domain-ontology-features | ::= | component-features. | (69) |

| | | | |
|---|---|---|---|
| model-schema | ::= | **<u>model-schema</u>** | (70) |
| | | conceptual-model-schema | (71) |
| | | verbal-model-schema | (72) |
| | | formal-model-schema | (73) |
| | | model-schema-features. | (74) |
| verbal-model-schema | ::= | verbal-definition. | (75) |
| formal-model-schema | ::= | formal-definition. | (76) |
| model-schema-features | ::= | component-features. | (77) |

case-model-schema ::= **case-model-schema** (78)
conceptual-case-model-schema (79)
verbal-case-model-schema (80)
formal-case-model-schema (81)
case-model-schema-features. (82)
verbal-case-model-schema ::= verbal-definition. (83)
formal-case-model-schema ::= formal-definition. (84)
case-model-schema-features ::= component-features. (85)

### A.1.3 Conceptual description of the components

#### A.1.3.1 Inference function

conceptual-inference-function ::= **conceptual-definition** (86)
name (87)
description (88)
input-roles (89)
output-role (90)
ontology-reference. (91)
input-roles ::= **input-roles** role+. (92)
output-role ::= **output-role** role. (93)
ontology-reference ::= **ontology-reference** text. (94)
role ::= **role** (95)
name (96)
description. (97)

conceptual-inference-structure ::= **conceptual-definition** (98)
name (99)
description (100)
inference-function-name+. (101)
inference-function-name ::= **inference-function-name** text. (102)

#### A.1.3.2 Control structure

conceptual-control-structure ::= **conceptual-definition** (103)
name (104)
description. (105)

#### A.1.3.3 Expansion method

conceptual-expansion-method ::= **conceptual-definition** (106)
name (107)
description (108)
inference-function-name (109)
inference-structure-name (110)
rational. (111)
inference-function-name ::= **inference-function-name** text. (112)

inference-structure-name ::= **inference-structure-name** text. (113)
rational ::= **rational** text. (114)

#### A.1.3.4 Control method

conceptual-control-method ::= **conceptual-definition** (115)
name (116)
description (117)
inference-structure-name (118)
control-structure-name (119)
rational. (120)
inference-structure-name ::= **inference-structure-name** text. (121)
control-structure-name ::= **control-structure-name** text. (122)
rational ::= **rational** text. (123)

#### A.1.3.5 (Role) Refinement method

conceptual-refinement-method ::= **conceptual-definition** (124)
name (125)
description (126)
input-inference-function-name (127)
output-inference-function-name (128)
rational. (129)
input-inference-function-name ::= **input-inference-function-name** text. (130)
output-inference-function-name ::= **output-inference-function-name** text. (131)
rational ::= **rational** text. (132)

#### A.1.3.6 Inference method

inference-method ::= **conceptual-definition** (133)
name (134)
description (135)
inference-function-name (136)
inference-name (137)
rational. (138)
inference-function-name ::= **inference-function-name** text. (139)
inference-name ::= **inference-name** text. (140)
rational ::= **rational** text. (141)

#### A.1.3.7 Domain Ontology

domain-ontology ::= **domain-ontology** entity-def⋆. (142)
entity-def ::= object-def | relation-def. (143)
object-def ::= atomic-object-def | constructed-object-def. (144)
atomic-object-def ::= concept-def | attribute-def. (145)

constructed-object-def ::= set-def. (146)
concept-def ::= **concept** *concept-name***;** (147)
[**sub-type-of:** *concept-name* ⟨**,** *concept-name* ⟩⋆**;**] (148)
[properties]. (149)
properties ::= **properties:** [property-def ⟨**,** property-def⟩⋆]. (150)
property-def ::= *property-name***:** value-set-def**;** (151)
[cardinality-def] (152)
[differentiation-def]. (153)
value-set-def ::= **number** | **integer** | **natural** | (154)
**string** | **boolean** | **universal** | (155)
**number-range(***number***,** *number***)** | (156)
**integer-range(***integer***,** *integer***)** | (157)
**{** *string-value* ⟨**,** *string-value*⟩⋆ **}**. (158)
cardinality-def ::= **cardinality:** [**min** natural] [**max** ⟨natural | **infinite**⟩]**;**. (159)
differentiation-def ::= **differentiation-of** *property-name***(***concept-name***);**. (160)
attribute-def ::= **attribute** *attribute-name***;** (161)
**value-set:** value-set-def**;** . (162)
set-def ::= **set** *set-name***;** (163)
[**sub-type-of:** *set-name***;**] (164)
**element-type:** object-ref (165)
[cardinality-def] (166)
[properties]. (167)
object-ref ::= ⟨ basic-ontology-object | domain-ontology-object ⟩. (168)
basic-ontology-object ::= object-type | (169)
**instance(**object-type**)**. (170)
object-type ::= **object** | **concept** | **attribute** | **set**. (171)
domain-ontology-object ::= domain-object-term | (172)
**instance(**domain-object-term**)**. (173)
domain-object-term ::= *concept-name* | (174)
*attribute-name* | (175)
*set-name*. (176)
relation-def ::= **relation** *relation-name***;** (177)
[**sub-type-of:** *relation-name***;**] (178)
[**inverse:** *relation-name***;**] (179)
**argument1:** argument-def (180)
**argument2:** argument-def (181)
[properties]. (182)
argument-def ::= ⟨ argument | **set(**argument**)** ⟩ **;** (183)
[**role:** *role-name***;**] (184)
[cardinality-def]. (185)
argument ::= object-ref | expression-argument. (186)
expression-argument ::= **expression(**object-ref**)** | (187)
**expression(***property-name* **of** object-ref**)**. (188)

#### A.1.3.8 Domain Model

domain-model ::= **domain-model** *domain-model-name***;** (189)
[**expressions:** text**;**]. (190)

#### A.1.3.9 Model Schema

model-schema ::= **model-schema** (191)
**model-ontology** (192)
model-term-def⋆ (193)
model-structure-def⋆. (194)
model-term-def ::= **model-term** model-term-name; (195)
**describes:** *domain-model-name* ⟨, *domain-model-name*⟩⋆; (196)
⟨ simple-term-def | complex-term-def ⟩. (197)
model-term-name ::= ⟨ *simple-name* | complex-term-name⟩ . (198)
complex-term-name ::= *complex-name*(*simple-name* ⟨, *simple-name*⟩⋆) . (199)
simple-term-def ::= ⟨ term-mapping | term-definition ⟩. (200)
term-mapping ::= **maps-to:** object-ref ⟨, object-ref⟩⋆;. (201)
term-definition ::= **defined-as:** ⟨ *simple-name* | **expression**(*simple-name*) ⟩;. (202)
complex-term-def ::= **maps-to:** *relation-name*(*simple-name*, *simple-name*);. (203)
model-structure-def ::= **model-structure** (204)
**describes:** *domain-model-name*; (205)
**parts:** part-def+ (206)
[**axioms:** text;]. (207)
part-def ::= *part-name*: part-element-def+. (208)
part-element-def ::= ⟨ single-element-def | set-element-def ⟩;. (209)
single-element-def ::= model-term-name. (210)
set-element-def ::= **set**(model-term-name) (211)
[cardinality-def]. (212)

# Appendix B

# Canonical Inference Functions

**Joost Breuker**

SWI, University of Amsterdam

This appendix describes canonical inference functions. It is an example of how a systematic analyses of a knowledge item (in this case an inference function) results in a complete set of library elements. Canonical inference functions are the result of a systematic analysis of inference functions based on a classification of the relation between input and output. They serve in the library as a typology of inference functions (basic modeling components). The idea of canonical inference function has been around already in KADS-I and this appendix recapitulates and expands on these initial ideas. In particular a new CIF, called select, is added which resolves an old debate about "specify". Another interesting aspect of this analysis is the way the ontology of input-output maps to the domain knowledge category. This treatment is conditional to a choice of representation in the domain category. Similar "exercises" can be done when different representation choices impose themselves (which is kept as a possibility in [Wielinga et al., 1992a]).

## B.1 Introduction

The typology of Canonical Inference Functions (CIF) is based upon a classification of the relation between the input and output dynamic roles. This can be paraphrased as: what is the difference between these parameters. These parameters stand for domain knowledge structures, therefore we need an *ontology* of these structures – as any KR formalism needs. Instead of constructing one we rather select an existing flavour.
Identifying in which respect an input knowledge structure differs from an output knowledge structure is dependent on the way the knowledge is represented. Therefore, there is no unique way to develop a typology of CIF. For instance, the input and output roles can be viewed as predicates over variables. In this format, only three major types of inferences exist: [1]

- a variable can be bound to a constant, or
- a predicate can lead to another predicate, or

[1] Another typology of inferences consists of the classical triad induction, deduction and abduction [Charniak and McDermott, 1985]. However, these are not functional types, but characterise inferences according to their control method.

- a predicate -in particular 'true' and 'false'- can be assigned as an argument to another predicate.

This schema is rather poor, and particularly for the transition between a predicate and another predicate it is difficult to find further meaningful refinements.
For the domain category description in CML there is already a pragmatic commitment to a terminological – i.e. KL-ONE like – knowledge representation (KR) model schema.
Another practical reason to have a terminological model schema in mind is that it has a rich expressive power. Its use results in a highly distinctive set of inference types.[2]
The default CIF typology proposed here resembles the one from the KADS–I Library. The CIF terms are based upon a model schema that has two basic structure terms: *descriptions* and *instances*. Descriptions are represented as terminological elements, i.e. as concepts and structures of concepts. Structures are composed of concepts and a special type of concepts which link concepts in a meaningful way: relations. Instances are represented as elements of *sets*. The link between the world of instances –i.e. the world of real events–, and the idealised world of descriptions is provided by the fact that concepts have instances, and that instances can be identified as occurrences of concepts. The former reflects the transition from idea to object or event; the latter the recognition of some object or event as belonging to a class (type). Note that "set–of–instance" has a different meaning than "class–of–instance". The typology that is derived is based upon the question "what can happen to a concept, respectively instance".
The elements on which inferences can be projected are: (see figure B.1).

- **structure (of concepts)**
  - **concept**
    - **attribute**
    - **value–restriction**
      - **value**
  - **relation**
- **set (of instances)**
  - **instance**

Figure B.1: Part-of hierarchy of CML domain terms to define types of CIS

Difference between this set of elements and those of the CML domain category are details on the use of quantification, which is not directly relevant here, and should only be applicable to sets; not to descriptions, and on the term *expression*. However, expressions are propositions that are composed of other CML domain terms (see e.g. figure B.2)
The terms concept, attribute, value(-restriction), relation, and instance are standard ones. A structure of concepts can also be a concept, which is the basis for expanding inference functions. Concepts are descriptions -models- of the world. They have a Platonic, i.e. intensional status. Instances are the interfaces between intensional descriptions and the 'real world' of individual events and objects. The act of recognising a specific object as an instance of a concept makes it possible to apply mental operations to that object. We attribute relationships to instances by 'imposing' or constructing a case model which covers the (relevant) instances on the basis of generic models, i.e. concepts.[3] Instances are individual elements, which *exist* or not [Hirst, 1989],

[2] Terminological representations have a strong object-oriented flavour, which makes them very suitable for constructing "generic models of systems in the world": see [Clancey, 1989], concluding section.
[3] This constructive view on perception and cognition is widely accepted in cognitive psychology. It does not mean that models can be arbitrarily applied to the events in the world: there should be some correspondence or 'reality principle'.

| | | |
|---|---|---|
| expression | ::= | descriptive state — <br> case state |
| descriptive state | ::= | concept relation concept — <br> concept attribute value–restriction |
| case state | ::= | instance attribute value — <br> instance $\in$ set |

Figure B.2: Expressions as propositions of CML domain terms

and which can be distributed over sets on the basis of similarities and distinctions among these elements. They do not have an 'inherent' organisation, like concepts have. Therefore, instances have no other structure than belonging to 'arbitrary' sets.
The distinction between concepts and instances is similar to the Fregean one between intension and extension (or: reference), which is also reflected in the T(-erminology) vs. the A(-ssertions) boxes in KRYPTON [Brachman et al., 1985] (see also [Costa et al., 1988] on definitions vs. assertions). Instances are the interfaces between these worlds. Hence, there should be operations to create or recognise those instances. The most typical ones are 'instantiate' which asserts the existence of a particular individual[4], and 'identify' which recognises a particular object or event as an instance of a concept.
We assume that there is a related distinction in making inferences *within* these worlds of instances and of descriptions. The operations in the T-box are different from those in the A-box in KRYPTON. Set operations apply to instances.[5].

## B.2 A classification of inferences

A classification of inferences can be derived from the question: "What can happen to a concept (instance)?", i.e. what are the effects of an inference in terms of conceptual structures.

**(1)** a concept can change, i.e. its values can change

**(2)** a concept can generate a new concept on the basis of a relation.

**(3)** two concepts can generate a relation on the basis of similarities and differences.

The (2) and (3) types of inferences can be viewed as solving equations. Given a concept and a relation, the related concept can be derived, i.e. found or generated; given two concepts, a relation can be derived, i.e. an identification of a type of difference (see figure B.3)
Figure B.3 presents a view of what an inference means in domain level terms. At the inference level, where roles stand for concepts, the type (2) inferences have one input and one output dynamic role and a supporting (static) role.[6] However, the type (3) inferences have two dynamic input roles and one dynamic output role; the latter is generally labeled as "discrepancy", or "difference".

[4] Instantiate only exists in a machine world which 'emulates' the real world. e.g. patient_4523 is an instance that has to be created and maintained. In the real world instances 'occur'. In phantasy worlds the distinction between occurrence and instantiation is hard to make.

[5] For that reason probability -or rather likelihoods- may exist in the world of instances; the occurrence of events can be sampled, generalised, etc. However, in the world of descriptions likelihoods make no sense. 'Uncertain' knowledge is a *contradictio in terminis*

[6] Supporting roles are "inputs" which are not affected by the inference function, but are used. For instance, abstracting data into evidence can performed by using a definition. Domain knowledge as represented by the knowledge base of a KBS plays always this supporting role. This is a static role. However, also dynamic supporting roles are possible, if, for instance, an inference function is performed on the basis of *derived* knowledge, i.e. the dynamic output role of some other inference function plays then the supportive role. This view is slightly different from what is stated in the 1.1. Deliverable (version 3); see also Appendix 1.

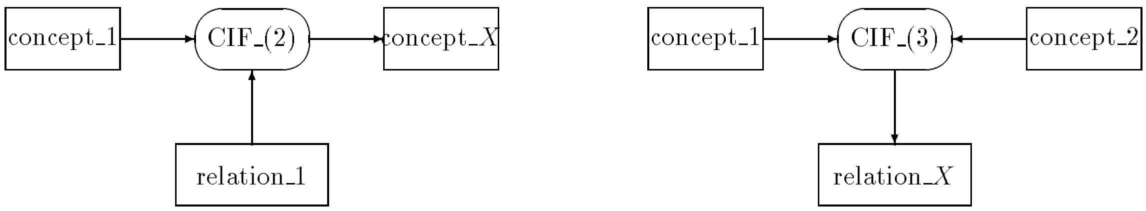



Figure B.3: *A schematic view on type (2) and type (3) inferences.*

A particular subtype of type (2) inferences is obtained when the concepts stand for structures of concepts. A structure of concepts is a concept, thus allowing for levels of grainsize in knowledge or expansion of inference functions. This subtype of inferences manipulates structures. It is debatable whether structure manipulation inferences are a distinctive type, or are in fact composed of type (2) inferences. For practical reasons structure manipulation inferences are handled as a separate type (4).
The fact that a concept can be expanded in a structure of concepts, implies that a CIF is not a *primitive* inference function. The typology of *canonical* inference functions has the property of being systematic in the sense of sufficiency and mutual exclusion, because it is based upon all combinations of elements as inputs and outputs.

**Change Concept: assigning values to attributes**

These type (1) CIF manipulate a value of an attribute of a concept.

**assign value**

*attribute, value restriction* → *attribute, value*

A ASSIGN VALUE CIF derives the value for an attribute, and assigns it. An example is the use of default values.

**compute**

*instances* → *attribute, value*

The input for an COMPUTE CIF is data, i.e. instances whose attributes have values, where one or more of the values is unknown. The output is an instance whose value is obtained. The domain knowledge which is used by this CIF may consist of a set of constraints, and the method employed can be for instance some constraint satisfaction regime.

**Generating new concepts, structures or instances**

The type (2) CIF which generate new elements are the most frequently used ones in inference structures.

**instantiate**

*concept* → *instance*

This type of inference function creates an instance of a generic concept. Instantiation involves assignment of values. In fact, in an instance *all* attributes should have values. These values may be given as data, inferred, or default ones.

**classify**

*instance* → *concept*

CLASSIFY is the inverse of INSTANTIATE. Another term may be: IDENTIFY, because this term may be easily confounded with the classification task (see Appendix C). CLASSIFY involves a method of comparing attributes of an instance with those that define a concept. The output is a concept which tells "what the instance is". The use of a flora to identify plants is an example of what is intended. Determining the nationality of a person is another one.

**generalise**

*set of instances → concept*

This CIF identifies one or more common properties (attributes) in a set of instances and tries either to map these on an existing concept, or to develop a new concept. The former resembles CLASSIFY, and classification methods may be part of the method description. The latter is generally known as *induction* as for instance employed in many machine learning methods.

**abstract**

*concept → (new) concept*

In the ABSTRACT CIF the relation between the input concept and the output concept is that the latter contains fewer attributes than the former one. Irrelevant attributes are abstracted (hidden; deleted). For instance, if one infers that a body temperature over $36^o$ C means that the patient has fever, this is called abstraction. What is relevant and what is irrelevant depends on a *point of view*. Points of view may be explicitly or implicitly prewired in the knowledge base (domain category) in the form of an abstraction hierarchy. A part_of hierarchy represents a structural point of view. A taxonomy -is_a hierarchy- reflects a functional perspective (rather than an ontological one). [Clancey, 1985], p 294, distinguishes some subclasses of abstraction on the basis of methods:

- definitional abstraction (e.g. "low WBC means leukopenia"), which we would rather call IDENTIFY.
- qualitative abstraction (the fever example above)
- generalisation (e.g. "a whale is a mammal")[7]

**specify**

*concept → concept*

SPECIFY is the inverse of ABSTRACT. A specification CIF produces a concept which has at least one more specific attribute than the input concept. A good example of specification is descending some taxonomy of concepts, i.e finding an appropriate subcategory. Another term for SPECIFY is REFINEMENT, although this term is often used to denote some complex trajectory of inferences that involve specifications.

**select**

*set of instances → instance*

SELECT has in the world of instances the same function as SPECIFY in the world of descriptions. An example of the use of SELECT is the selection of a hypothesis from a set of hypotheses –if this set of hypotheses is a real set, and not a structured set. The SELECT inference function often needs an explicit support role to state on the basis of what data something is selected. This support role is often a dynamic one, i.e. the dynamic output role of another function.

**Two concepts and their relation**

These type (3) CIFs produce a concept that expresses what is common or what is different between two concepts. The output is some difference. These CIFs are often used to infer whether goals are obtained in reasoning, and can easily used for control purposes, i.e. the output role can be the support role for some other CIF, in particular a SELECT CIF. MATCH and COMPARE CIF connect various lines of inference.

[7] In the literature the term generalisation is often used for both of what is here described as abstract and as generalise. Generalisation is performed over sets of instances, in the same way as it is applied in inferential statistics. In our typology Clancey's generalisation is typical an abstraction over concepts.

**compare**

*value of X, value of Y → concept with difference value*

The values of (the attributes of) concepts are compared. The concepts are input for the CIF. The output is a class (equal, not equal) or a difference value.

**match**

*structure X, structure Y → concept with difference structure*

In its basic form a MATCH CIF has two structures as its input. These structures are compared. The output is a description of where the two structures are different, or its inverse: where the two structures overlap. Like COMPARE, the function of MATCH in many inference structures is to provide feed-back, or other forms of content dependent control, and can be used to focus for further refinements. The description of the difference can be an existing concept. An example of MATCH is the use of metaphors or models, where the output is a description of the structural *overlap*.

**Structure Manipulation**

These inference function manipulate structures by changing them into some other structure. In principle there can be as many structure manipulation CIF as types of structures can be distinguished. Here we will present only some frequently occurring ones:

**assemble**

set of instances (components) → part_of structure

All ASSEMBLE inferences take as their input a number of elements and produce a part_of structure. ASSEMBLE is an important CIF in synthesis tasks like configure, planning, design, etc. The methods for assemble, e.g. constraint satisfaction may be very complex, which means that an ASSEMBLE CIF is in practice often decomposed in many other CIF, leading to a more simple version. A good example of the role of ASSEMBLE is the SALT knowledge level architecture for configuration tasks [Marcus, 1988].

**decompose**

part_of structure → set of instances

DECOMPOSE is the inverse of ASSEMBLE. The input is some composite structure and the output is a set of components. An example of a DECOMPOSE CIF is the breaking down of a global, or composite hypothesis into a set of more detailed hypotheses. In this respect, DECOMPOSE can have a similar, but more general effect than SPECIFY.

**transform**

*structure1 → structure2*

TRANSFORM converts an input structure into another structure. Essentially, there are two types of TRANSFORM. The first type leaves the structural elements identical, but simply reorders elements within a structure (e.g. sorting procedures). Another, more interesting type of transformation is where new structural descriptions are assigned to the elements. For instance, a sequence of elements can be transformed into a part-of structure (parsing). Transformation CIF may in often hide compounds of other CIF, and appear in conceptual models only at very high levels of description.

Other structure manipulation CIF can be defined. For instance, SORT, which changes a set into a sequence, or PARSE, which assigns a part_of structure to a sequence.

In figure B.4 the typology of CIF is summarised.

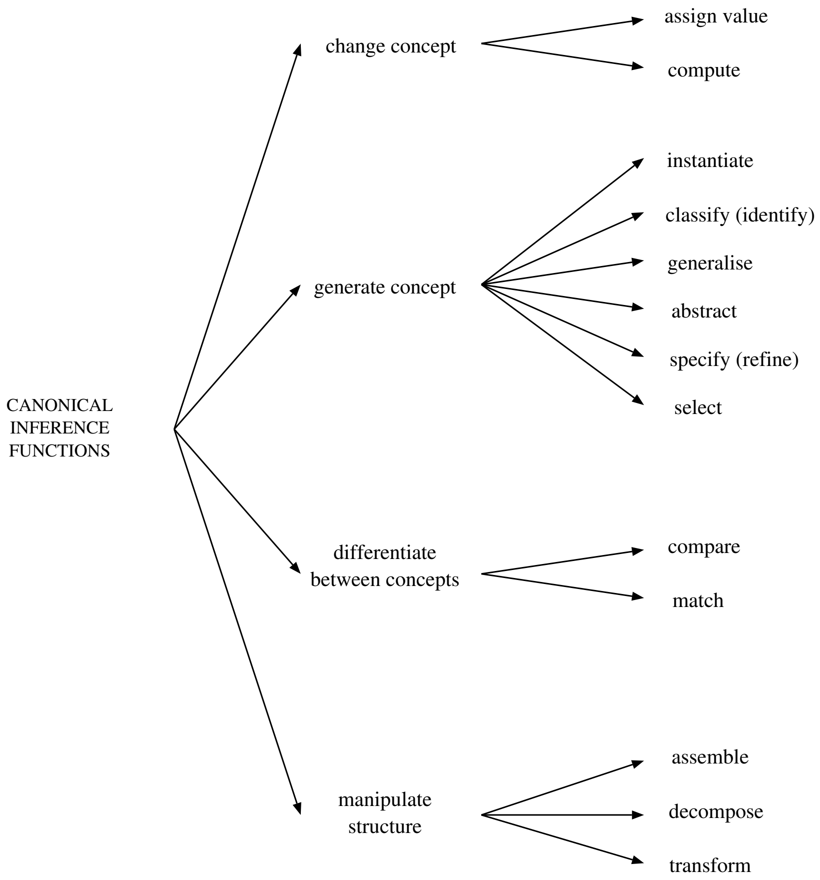



Figure B.4: Typology of Canonical Inference Functions

# Appendix C

# Basic Tasks

**Joost Breuker**

SWI, University of Amsterdam

This appendix describes a typology of inference functions, the basic tasks. Basic tasks are the result of an analysis of taskss in terms of what can be done to a system. This leads to classical distinctions between, for example, analysis and synthesis tasks. The basic tasks form a taxonomy that can be used as a decision tree to classify a task at hand. Based on the result of this classification further modeling decisions can be taken. The taxonomy of basic tasks is an important part of the library. It is expected that it can be integrated into the library framework without much difficulty.

## C.1 Introduction

Following [Clancey, 1985] the major subdivision of Basic Tasks is based upon the operations that can be performed to a (model of a) system-in-the-world. The system – e.g. a human body, a bank, a device etc. – or its components or attributes can be identified or classified. These are called analytic tasks. Analytic tasks are characterised by the fact that the solution for a problem is part of the domain knowledge. In diagnosing diseases, the diseases are known, and the task consists of identifying which of these is applicable to a patient. In troubleshooting the components of the device and their (correct) functioning are known. This is in fact a short–cut description of what an analytic task is in terms of a system–model. The complete picture is that analytic problem solving consists of the construction of a case model on the basis of a system model. Diagnosis is than the construction of a model where a faulty component takes the role of a correctly functioning one, or where a malfunction is derived from a model of e.g. physiological processes in which disruptive processes influence the functional ones. In other words, a principled analytic problem solving process implies the construction of a coherent and 'complete' (but abstracted) model of the case. The short–cuts are based upon the fact that parts or processes can be isolated from the system and can be *classified*. For instance, the identification of a fault class, such as diseases, may be the intended outcome of a diagnosis, thus reducing the whole model–construction process to the classification of a case. The goal of analytic tasks can therefore be stated as the establishment of an unknown property or component of the system by classifying its behaviour (data). The typical problem in analytic tasks is the question: *"what is the case with a system?"*

In synthesis tasks this is not the possible. In synthesis –or "construction"– tasks the system is not given, but has to be constructed from elements given, or assumed, and a specification of the required behaviour (functions). In a synthesis task the goal is to assemble a structural description of a system in terms of known parts.
KADS–1 defined an intermediary category of basic tasks: system modification tasks. These tasks are "small" construction tasks, in which the structure or functionality of a system is maintained, but a component, or process is modified, e.g. for repairing the system.
Note that all these operations are symbolic representations of e.g. real physical actions, and that there may not be a one to one mapping between the problem solving "emulation" and the real thing. Constructing the design of a house and actually building the house do not easily map onto one–another. Even the mental activities involved may largely differ. The latter task involves predominantly the planning of physical activities; the former is in the first place a spatial design task.

## C.2 Analysis Tasks

In the KADS–I Library of Interpretation Models "generic analysis tasks" were relatively well developed. Most of the tasks distinguished there will also figure in the CommonKADS Library of Re–Usable Models. More specifically, these are:

**Classification:** Classification is implied in all of the next tasks. They are called classification tasks, because the the solution of a case consists of attributing a class to a case, i.e. classifying a case. Simple classification would imply no more than collecting values for attributes, and matching this to the class descriptions. However, this very abstract characterisation of the most well known analysis tasks misses out what is required to happen "in between". A simple (e.g. heuristic) match will only do in the simplest or simplified cases. What all these basic tasks have in common is that they have some method of *testing* involved in assessing whether a case fits a class.

- **Monitoring:** In monotoring there are only two types of classes: whether there is a discrepancy or not between an expected, or norm value, and an observed value. A discrepancy is an *error*, or a *complaint*. [?] call this type of task: *verification*, and its family resemblance with diagnosis is so strong, that they point out that the use of "diagnostics" should not be confused with real diagnosis, i.e. the tracing of the cause of discrepancies.
- **Diagnosis:** Diagnosis is probably the most investigated basic task, because of its high practical value. Any Time a system does not behave according to our expectations or norms, a search for some *causal* factor is required if we want to repair or remedy (i.e. modify) the system. The causal factor may be a defective component or a disruptive process. What is considered to be a cause is dependent on epistemological and pragmatic views in the domain. Whatever they are, in diagnostic tasks these causes are (fault) classes to which the malfunction can be attributed. This means that there may be many classes, and the classes may not be enumerable, but they should be definable to allow tractable diagnosis.
- **Assessment:** Assessment resembles monitoring because it also knows two classes: whether a case is "accepted" or not. However, it differs fro monitoring because the case is not preprocessed in such a way that it can be interpreted in these terms. A typical example is deciding about a legal case. the case consists of some story of events, only some of which may be relevant for the decision.

**Identification or prediction** In identification the case is not classified, but the value of an attribute is derived from an input specification of the case. A specific value may be aimed at, or a (complete) behavioural description of a system may be the solution.

## C.3 Synthesis Tasks

The synthesis tasks in the KADS–I Library of Interpretation Models were far less well developed than the analysis ones. Our understanding of these tasks is far less well developed as the (AI, Cognitive Science) literature easily shows. In the CommonKADS Library we will identify more or less the same basic tasks, but they are organised differently from the old library, because of new insights.

In synthesis tasks, understanding by way of tracing of what is the case with a system, only plays a minor, evaluative role, because there is no system yet. It has to be constructed on the basis of specifications of its prospective behaviour. The solutions are not known, only the potential elements or structures of the solution. Therefore, construction tasks are more complex than analysis ones, if only because the solutions consist of combinations instead of single elements.

If viewed in this way, construction tasks are almost impossible in a blind generate and test framework. A blind generator would first propose a combination, next its behaviour under the required circumstances (input states, or initial states) would be derived, and the behaviour would be tested to find out whether it fits the functional requirements or not. To tame this combinatorial explosion there are two classical ways out (see also [?]). The first one is by using abstraction layers, in the same way as in Common–KADS for constructing KBS. The use of modelling languages is a typical example. The second one is the "divide–and–conquer" principle in which (the requirements for) the solutions are decomposed. There is also a third way out, and that is by the use of abstracted solutions to typical synthesis problems: these are often called *skeletal* solutions. This CommonKADS Library is an example of the (re–)use of skeletal solutions. Synthesis problem solving driven by skeletal solutions is often called "routine" design. In routine design the problem of generating potential solutions is reduced to matching a specification to a set of abstract solutions, i.e. the synthesis task is *degraded* to an analytic task. Most well known methods in AI to solve synthesis problems are routine ones (e.g. [Brown and Chandrasekaran, 1989; Marcus, 1988; Friedland and Iwasaki, 1985]).

Not only the available support knowledge –i.e. modelling languages, solution decompositions and skeletal solutions–, but also the task *features* can make a construction task more or less complex, because parts, or features of the solution are already given. [Kruger, 1990] classifies construction tasks according to which elements and/or structures are 'given' in the input (see figure C.1.)

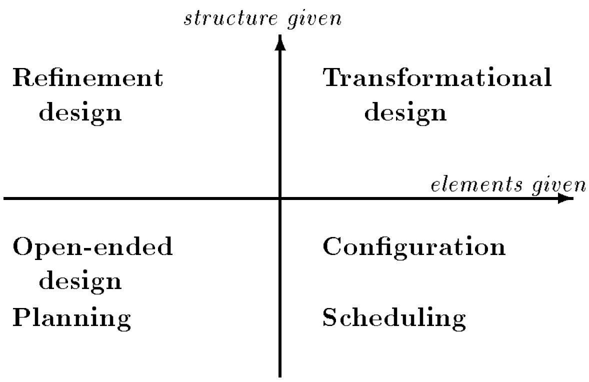



Figure C.1: Classification of synthesis tasks according to what is given.

Synthesis tasks are more heterogeneous than analysis tasks. Despite these large differences it turns out that the global task structure for synthesis tasks is the same. Invariably empirical investigations show an analysis stage, in which the problem specification is classified, or transformed in order to allow decomposition of the problem, which is followed by an assembly (synthesis) of the solutions.

This is essentially also what [Descartes, 1637]'s famous method prescribed. This pattern may be repeated per stage when intermediary constructions can be made.
This stereotypical method for solving synthesis problems, which hides so much heterogeneity, suggests that task, or inference structures are of little 'guiding' importance, and rather an epiphenomenon of applying *domain dependent methods*. If that is the case, the Domain–Division of the CommonKADS Library should provide the guiding principles. Indeed, most complex synthesis tasks develop domain dependent methodologies, resulting in engineering disciplines. Knowledge engineering is one of these.

## C.4 Modification Tasks

There are two basic system modification tasks:

**Repair** In repair tasks the solution is the replacement of a component by another component, which has the same structural properties, i.e. it should "fit" the structure. In general one wants also to preserve the functional integrity of the system, but it is also possible that the repair enables additional functionality (e.g. replacing a computer board by a more powerful one). In general repair tasks are contingent upon diagnosis of defective components. Repair tasks are in general rather straightforward. Complications are the generation of correct (dis–)assembly plans to install the new component.

**Remedy** Remedy tasks may be more complex. In remedy a process that behaves incorrectly, i.e. a malfunction, is counteracted by starting up another process. Remedies may not only contingent upon the diagnosis of malfunctions, but are particularly relevant in correcting errors, i.e. they can provide the feedback of a monitoring task. Remedies are contingent on monitoring tasks in correcting errors of a system, thus providing a process feed–back control loop. Remedies contingent upon monitoring may also have the appearance of "suppressing symptoms" and may not be a long lasting solution. However, remedies can also be used to maintain the functional integrity of a system when the cause of a disturbed process has been identified as for instance an interfering process, i.e. after diagnosis. In "killing" such a process, the remedy can be well founded. The use of remedies is typical for complex systems, e.g. in medicine.

Summarising: the access to the Task–Division occurs by assessing the *features* which characterise the solution. Therefore, the taxonomy of basic tasks may also be called a taxonomy of types of problems. As in any taxonomy, these features are distinctive features, the taxonomy can also be read as a well structured decision tree. In figure C.2 the taxonomy of basic tasks is presented.

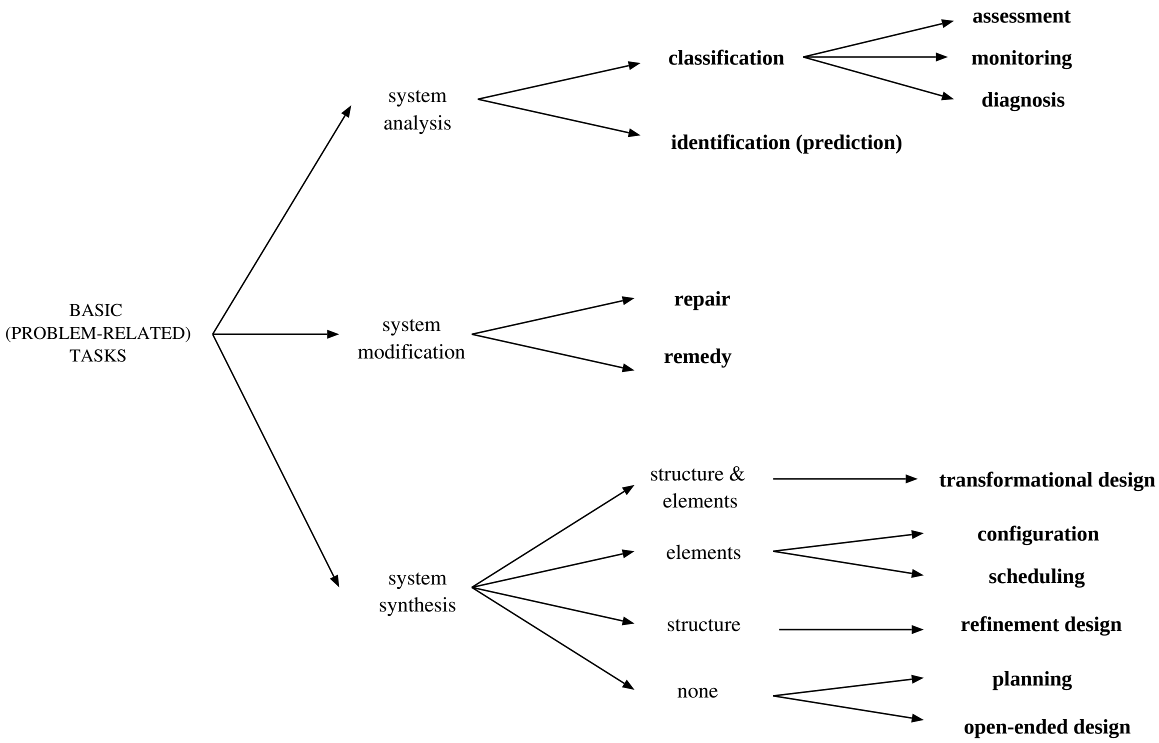



Figure C.2: Taxonomy of Basic Tasks.

# Appendix D

# Role Limiting Methods

**Cuno Duursma**

VUB AI-Lab

This appendix describes three role limiting methods; Cover & differentiate, Propose & refine and 'extrapolate from a similar case'. I have presented these RLM's independently from their usual field of application and have included separately applications on diagnosis, classification, assessment and design tasks. The RLM's are described by a detailed specification of their domain organization, task structure and inference structure. It has been tried to make the descriptions closely reflect the *Common*Kads framework under development, but they can also be read and understood without this framework.[1]

The discussion in this appendix is of interest to the library for a number of reasons. First it provides interesting examples of generic models. Second it provides interesting ideas about operations on generic models. In particular the way in which the application of a single role-limiting method to different tasks is being described offers some clarifications on items of problem solving knowledge. Problem solving knowledge does not only consist of the methods, but also of knowledge about how to apply these methods to a task. The "terminology translation tables" described in this appendix are examples of this kind of knowledge.

## D.1 Introduction

Role Limiting Methods were introduced in [McDermott, 1988]. Role limiting methods reduce the roles of knowledge in problem solving. In this way they both support knowledge acquisition, knowledge representation and efficient and seperately represented problem solving control. Role limiting methods require a certain domain model organization and provide control mechanisms to apply the domain knowledge to make certain types of inferences in order to reach an end state. RLM's should not be confused with weak methods or problem solving methods.

Weak methods have weaker domain requirements, and have therefore a wider range of applicability. One problem with weak methods is that the domain ontology is not limited; which causes irregular representations. A second problem is that search control is often not sufficiently powerful so that

[1] This appendix was previously made available as document KADS-II/T1.3.5/VUB/CD/0.2 (Draft version 2; Refer to as **VUB AI Lab Memo 92/4**)

the domain knowledge has to be extended with problem solving control [Duursma, 1992a; Laird et al., 1986].
A problem solving method is in KADS-II is described as "a description of how to achieve a certain (sub-) task, characterized by a goal and input/output specifications, through the generations of subtasks" [Wielinga et al., 1992a]. A RLM extends this concept with an exact description of the domain operations to be supplied.
In this report we have described the RLM's therefore by a their domain organization, their task decompositions, inference structure and control.

- domain organization
  - domain model vocabulary
  - domain model ontology requirements & assumptions.
  - domain operations
  - case models
  - domain representation
- task structure
  - task decomposition
  - task control
- inference structure

A single RLM is the combination of descriptions in these three categories.
In [Marcus, 1988] five RLM's are discussed; cover and differentiate, 'qualitative reasoning and cover and differentiate', propose and revise, acquire and present and extrapolate form a similar case. Of these only C&D, P&R and the extrapolations approach are complete descriptions and independent of a specific task.
In [Duursma, 1992a] a RLM called structural focusing has been identified. This RLM has been refined and renamed to 'hierarchical focusing' and will be shown in this report (in the next version, sorry).
RLM's are not specific for a particular class of tasks. They are applicable on any task within an application domain where the domain assumptions of the RLM can be satisfied. This makes the RLM in practice applicable on a range of tasks with similar characteristics. A RLM that is easily applicable on diagnosis can not be used easily for design, because these two tasks have very different characteristics. In the descriptions of the individual RLM's we have provided a vocabulary that is most natural for the RLM and vocabulary matchings for a set of applicable tasks. This vocabulary is used in the labeling of the knowledge roles and in the rephrasing of the domain model requirements. The knowledge roles form a subset of the vocabulary.

## D.2 Cover & differentiate

Here we discuss the original cover and differentiate approach [McDermott, 1988; Eshelman, 1988]. A role limiting method can be used for any task where the domain model assumptions hold. So not only for diagnosis, but also for other basic tasks, like planning, scheduling, assessment, design. Also the task control is not necessarily task specific.
For the different tasks the vocabulary is investigated. This results in a relabeling of the knowledge roles. Otherwise it is hard to understand what the knowledge roles, inferences and assumptions mean within the context of the task.

### D.2.1 Domain Knowledge

#### Vocabulary

- abnormality: same as event.
- associated symptoms: The symptoms associated with an explanation.
- cause: An event that explains the symptoms and for which no explanatory connections exist (i.e. the end of an explanation chain).
- cover: A set of explanations that is associated with a set of symptoms (a cover is not strong enough to be a justification).
- differential: The difference between an explanation on one hand, and a set of required symptoms (symptoms + value) and a sub-set of explanations on the other hand.
- environment: The world the system interacts with.
- event: A problem with the structure, sub-function, configuration or state of the system that produces symptoms.
- enabling events: Events that are the precondition for the applicability of a explanatory connection or explanatory event.
- explanatory event: A stored association between a set of symptoms and an event, it explains a set of symptoms.
- explanatory connection: A stored association between an event A and an event B, where A has an 'explanatory relation' with B.
- initial symptoms: The initial symptoms the approach starts with.
- intermediate symptoms: The symptoms that are required to differentiate possible explanatory events.
- justification: A justification for an explanation is path starting from symptoms to a set of events via explanatory events and from these events to other events via explanatory connections, where each of the required symptoms of the events are available while there is no more specific event than the explanation of which all the required symptoms and enabling hold.
- problem state: same as event.
- required symptoms: The symptoms required by an explanatory event or explanatory connection to be valid (also called 'anticipatory knowledge' [?][55]).
- symptom: An output produced by deviational behaviour (event) of a system. Also: an observable event.
- system: A structure that interacts with an environment and produces observables.

#### Domain model ontology requirements

- 'There is an identifiable set of problem states or events.' (original) This means that all problems states that this approach is going to recognize should be listed exhaustively.
- There is a pre-enumeration of possible symptoms.
- 'Statically pre-enumeration of candidate explanations that cover events'. (original) This means that the covering operators can be listed for all events.
- 'There is information to differentiate the candidate explanations for each event' (original). This means that there should be sufficient symptoms observable and available to separate the different covering explanations. This is a requirements both on the events and on the type of explanations.
- 'One candidate explanation is applicable at any given time per event' (the single fault assumption, original).

### Case Models

The Case model represents here the states that the problem solver will go through while working on the problem.

- In C&D the intermediate case models (states) represent explanations that cover the initial symptoms. The explanations have different levels of justification by the available symptoms (i.e. their required or enabling symptoms are not all requested or available) and have different depth of explanatory connections.
- The initial case model (start state) represents the explanation that covers the the initial symptoms.
- The solution case model (end state) represents an explanation that fulfills the initial requirements and intermediate constraints and has no explanatory connections.
- The 'case user input data' is a set of symptoms necessary to differentiate the covering explanations.

### Domain operations

The Domain Model for C&D contains the following types of information:

- Symptom, associated event pairs. 'The form of each piece of knowledge must be such that touching a symptom returns the set of associated events' (original).
- Event, differentiation information pairs. 'Touching a candidate explanatory event returns a description of the information that will help to confirm or disconfirm the event'. (original) This is actually in a more detailed form:
  - Events are classified into testable (symptoms) and non-testable events.
  - Events are associated with required events.
  - Events are associated with events that enable an explanatory connection or an explanatory event.

The symptom, events pairs are quite clear and can be found in a domain where the domain assumptions hold.
Differentiation support contains observable events (symptoms), the required events and the associated events.
Another piece of information is the order in which observations should be made. In most domains the cost of each of the observations can be different and there is a preferred order. This information seems not to be represented explicitly in the standard C&D RLM.
The following operations are directly supported by the domain model:

1. *cover-op(+S:symptom, -El:event*).*

   Associates a symptom with a set of events. The cover operation handles one symptom only. For covering a set of symptoms a (small) inference has to be made to collect the events of each of the separate symptoms.
2. *testable-op(+E:event, -Testable:bool)*

   Returns true if the event is testable (i.e. a symptom) otherwise false.
3. *required_events-op(+E:event, -Re:event*)*

   Returns the list of required events of an event.
4. *enabling_events-op(+E:event, -Ee:event*)*

   Returns the list Ee of enabling events of event E.

**Domain model representation**

Here the graph with uni-directional links should be described.

### D.2.2 Tasks

The top level task of C&D can be considered to build up a theory that prescribes the observations to be made to conclude a set (event is set of symptoms) in a set-subset structure. The top level approach of C&D is to find this theory by searching through a set of existing ones. The existing theories are indexed by a limited set of values of different, observable events. Following this strategy, the first decomposition is to divide the task into a covering step and a differentiate step. Cover takes a set of features (symptoms) and produces a set of theories (explanations) that seem applicable; differentiate tries to rule out elements of this set. The cover and the differentiate sub-tasks are not equally complex; the domain ontology selected in [?] directly supports the cover task by a cover domain operation. Cover can therefore also be classified as a kads-II 'primitive-inference-action'[2] (PIA). The differentiate task is not directly supported by the domain ontology. It is therefore necessary to decompose this task.
Sub-tasks of differentiate in the C&D approach are:

1. differ-task(+E:explanation*, +IS:symptom*, -R:request*). This task takes the explanations and searches for a set of differentiating symptoms. These symptoms are then requested to the test task. There are four different ways to do this;
   - By directly testable competing events. Some events are directly observable and can be passed directly as a request to the test sub-task.
   - By finding symptoms that rule out competing events. Different events have always some non-overlapping symptoms, testing these rule out at least one of the competing events.
   - By finding symptoms that form an enabling connection. Some competing events have associated enabling events. Is the enabling events cannot be shown, then the competing event cannot be an explanation.
   - By using required symptoms of competing events. If the required event is not available then the competing event cannot be an explanation.
2. test-task(+R:request, -S:symptom). The test task checks whether a requested symptoms available.
3. infer-task(+E:explanation*, +Os:symptom*, +Ns:symptom*, -NE:explanation). The infer task takes the set of competing explanations, the original symptoms, the new symptom requested by the differ task and checked by the test task. Using this information infer tries to eliminate competing explanations and produces a new list of explanations (NE). There are different ways to do this:
   - If the requested symptom was not available and is as an 'testable event' part of the competing events, then this competing event can be eliminated.
   - If a competing event does not cover the new symptom, then this event can be eliminated.
   - If the requested symptom was not available then the events that have this symptom as enabling connection can be removed.
   - If the requested symptom was not available then the events that have this symptom as required symptom can be eliminated.

[2] In [?] it is called a knowledge source

**Task Control**

Task control can be specified for each task.
Control specifies:

- The order of the subtasks.
- The number of executions of each sub-task.
- The arguments on which the tasks are applied

### D.2.3 Inference structure

The inference structure links inferences and knowledge roles. Figure D.1 shows a visualization of an inference structure for the C&D approach.
It appears that inference relations exist between primitive inference actions but also between other tasks. Here we present inference relations as required relations between the input and output of tasks or primitive inference actions. Thus we represent what kind of relations can be created, but not how they are achieved. The way of achieving is described in some detail in the task decomposition (which can be generalized in to the problem solving method). (also D1.1 issue) Therefore inference structures have no control and thus no order.
Inference structures can in principle be defined at each level of task decomposition. We do not know yet whether it is useful in practice to do this. Figure D.1 shows an inference structure at the first level of decomposition.
All inference relations should be defined in terms of other inferences or grounded on domain operations. Every inference relation should bottom out on domain relations.

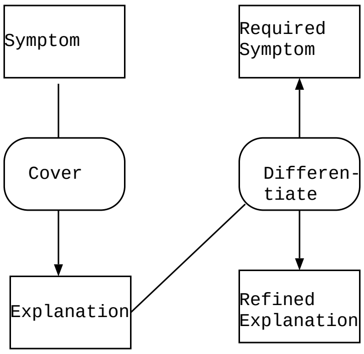



Figure D.1: Inference structure of Cover and differentiate at first level of decomposition.

1. *cover-ir(Symptom, Explanation)*

   Cover is a binary inference relation between a set of symptoms S and a set of explanations E. Cover is not reflexive, it is distributive. We define the cover inference relation as follows:

   There is a binary relation R(symptom, explanation): Cover(S, E) holds when for each of the elements s of S and for each of the elements e of E R(s, e) holds.

2. *differentiate-ir(Symptoms, Explanations, New symptoms, Refined Explanations)*

   Differentiate is an inference relation between a set S (the Symptoms), a set E (the original Explanations), a set N (New symptoms), and a set F(the Refined Explanations). Differentiate is not reflexive, it is distributive. We define the differentiate inference relation as follows: Differentiate(S, E, R, F) holds If

   - cover(S, E) holds

- cover(F, E) holds
- E a subset is of F
- If the result of the observation request is positive then should F cover N too.
- If the result of the observation request is negative then R may not be apart of either the required symptoms or the enabling symptom of an element of F.

In the same way the inference relations differ, test and infer can be defined.

### D.2.4 Dataflow of C&D

I have added a dataflow for this approach in figure D.2. This is currently not part of D1.1. The dataflow represents the flow of information as presented in [McDermott, 1988]. In contrast with the inference structure the arrows actually represent flow of information and not relations between data-types.
This can be compared with the difference between a relation table of a relational database (i.e. the inference structure) and a dataflow of how an actual query to the database manager is handled (the dataflow). The inference structure can also be compared with a numeric relations (e.g. $x \geq y$), while the data-flow can be compared with numeric operations (e.g. $x + y$). It can be argued that this type of dataflow belongs to the *Common*Kads *design model*.
The squares represent data of certain type, and the ovals represent processes (operators). The complete process starts by reading in the initial symptoms and selecting on that basis applicable symptom-explanation pairs from a store of available, pre-enumerated symptom-explanation pairs. Then a selection from the applicable symptom-explanation pairs is made to focus on. This is a complicated process (and this box hides this complexity) in which all kind of rational and observational constraints should be taken into account. For the resulting pairs the required symptoms are retrieved. Observations is requested for these symptoms. This results in an update of the symptom list and the whole process is repeated.

### D.2.5 C&D applied on Diagnosis

**Task vocabulary**

| Cover and differentiate | Diagnosis |
|---|---|
| event | fault, malfunction, component + state configuration. |
| explanatory connection | explanation, cause |
| explanatory event | explanation, cause |
| initial symptoms | complaints |
| matching explanation | idem |
| possible explanation | idem |
| possible symptom | idem |
| required symptom | idem |
| symptom | idem |
| system | device |

**C&D assumptions rephrased for diagnosis**

- There is an identifiable and enumerable set of faults.
- There is an identifiable and enumerable set of symptoms.
- There is a static enumeration of symptom, explanation pairs.

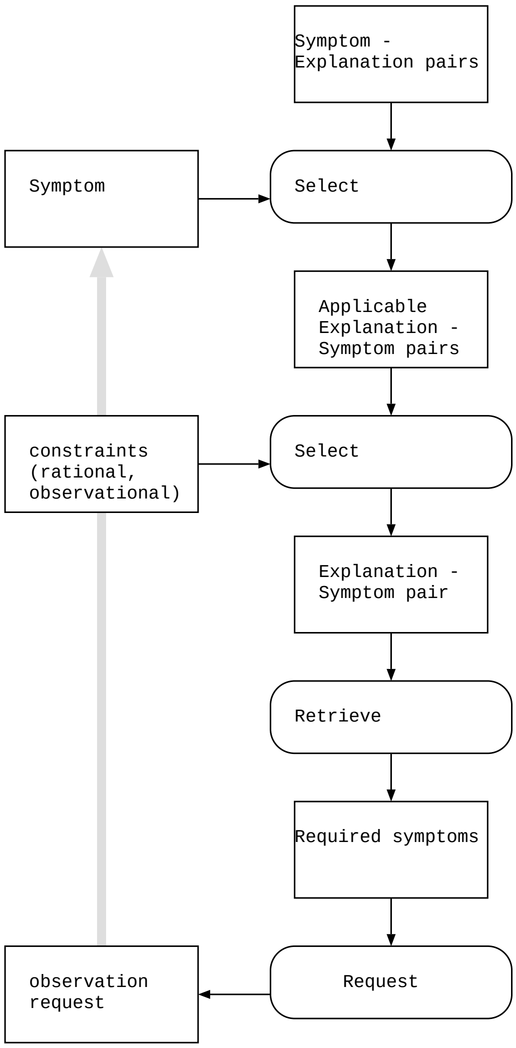



Figure D.2: Data flow of Cover and differentiate.

- Sufficient case symptoms are available and observable to differentiate the possible faults.

- There is only one fault at any time given.

### D.2.6 C&D applied on Classification

The task of classification has the goal to find the most specific class of a class taxonomy that fits on a specific instance. The instance is specified by a set of initial features. On the basis of these features an initial set of applicable classes is selected. These classes are than differentiated by finding features that separate these classes. Than more specific classes are searched for. When no more specific classes are found the process stops and the most specific class that has all its required features matching with the instance is returned.

**Task vocabulary**

| Cover and differentiate | Classification |
|---|---|
| event | class |
| explanatory connection | class, sub-class connection |
| explanatory event | class, features connection |
| system | taxonomy |
| symptom | class feature |
| problem state | instance (?) |
| possible explanation | possible class |
| matching explanation | matching class |

**C&D assumptions rephrased for Classification**

- There is a identifiable and enumerable set of classes.
- There is a identifiable and enumerable set of instance features.
- There is a static enumeration of class, feature pairs.
- Sufficient instance features are available and observable to differentiate the possible classes.
- An instance belongs to one class only at any time given.

### D.2.7 C&D applied on Case based Assessment

The task of case based assessment has the goal to find the most specific decision of a decision taxonomy that fits a specific case. Case based assessment can thus be viewed as a specialization of classification.

**Task vocabulary**

| Cover and differentiate | Assessment |
|---|---|
| event | decision |
| explanatory connection | decision, more specific decision connection |
| explanatory event | decision, case features connection |
| system | measurement system |
| symptom | case feature, observable |
| problem state | case (?) |
| possible explanation | possible decision |
| matching explanation | matching decision |

**C&D assumptions rephrased for Assessment**

- There is a identifiable and enumerable set of decisions.
- There is a identifiable and enumerable set of case features.
- There is a static enumeration of decision, case feature pairs.
- Sufficient case features are available and observable to differentiate the possible decisions.
- A case belongs to one decision at any time given.

### D.2.8 C&D applied on Design

**Task vocabulary**

| Cover and differentiate | Design |
|---|---|
| event | design |
| explanatory connection | design, more<br>specific design connection |
| system | |
| symptom | requirement, constraint |
| problem state | requirement |
| possible explanation | possible design |
| matching explanation | matching design |

**C&D assumptions rephrased for Design**

- There is a identifiable and enumerable set of designs.
- There is a identifiable and enumerable set of case requirements and constraints.
- There is a static enumeration of design, constraint, requirement triples.
- Sufficient requirements and constraints are available and observable to differentiate the possible design alternatives.
- A case belongs to one design at any time given.

## D.3 Propose & revise

Here we discuss the original Propose and Revise role limiting method used in the SALT shell [Marcus, 1988]. P&R is traditionally applied on configuration, but can in principle also be used also for other basic tasks, like planning, scheduling, assessment, diagnosis. Also the task control is not necessarily task specific.
For the different tasks a renaming of the domain model vocabulary has been provided, otherwise it is hard to understand what the operators and inferences mean within the task. (D1.1 issue).
Propose & revise looks very similar to Cover & differentiate. There are however differences.
A different domain organization is used in P&R. C&D uses a number of pre-stored solution classes (explanations) in which the problem instance is classified. P&R gradually builds up a solution by using pre-stored partial, interacting solutions. These are pairs of a single requirement or constraints associated with the value of a single or a few dimension dimensions. The result is that domain organization tasks and inference relations are different.

### D.3.1 Domain Knowledge

**Vocabulary**

- antagonistic constraints: Set of constraints that are strongly interfering.
- constraint weight: The importance of satisfying a constraint.
- constraint: A non-functional requirement (this is however not an absolute classification).
- design extension: An addition to a design specification, in order to fulfill more requirements.
- design: A specification of a structure.
- evaluation: A test that shows if -or in what extend- a design satisfies the requirements.
- fixed constraint: A constraint that may not be violated.

- partial design: An incomplete specification of a structure.
- proposition: A design that has not yet been evaluated.
- remedy: A modification to a design in order to take new constraints into account.
- requirement: A functional constraint to the design.
- revision: A constraint-violation remedy pair.
- space: The possible solutions to satisfy constraints and requirements.
- thrashing: Search in different parts of the space keeps failing for the same reasons [Gasching, 1979], caused by node-inconsistency or the lack of arc consistency [Kumar, 1992].

**Domain model ontology requirements**

- Design can be specified by values for a fixed number of attributes.
- 'Procedures can be specified to determine the starting attributes and their starting values, most likely values and most preferred values' (original).
- There is an enumerable set of design constraints.
- 'For each design constraint remedies in terms of modifications to the attribute values of the design can be specified' (original).
- 'There is not a high level of potential conflict for alternative design extensions' (original).

**Case models**

- In P&R the intermediate models (problem solving states) represent versions of the design that cover in more or less extend requirements and constraints.
- The initial case model (start state) represents a design that covers the (initial) requirements.
- The solution case model (end state) represents a version of the design that satisfies both the requirements and the constraints.

**Domain operations**

The domain model of P&R contains the following types of information

- Attribute, value constraint pairs.
- Attribute, proposed value algorithm (procedure) pairs.
- Constraint violation, revision pairs.

The following relations are directly supported by the domain model:

1. *constraint-op(+A:attribute, -C:constraint*)*

   Returns a list of constraints on the value of the attribute. These constraints can either be absolute or in terms of the values of other attributes.

2. *propose-op(+A:attribute, -V:value*)*

   Returns a value (or a list if the attribute is multidimensional) for the attribute. The value is either absolute or in terms of the values of other attributes.

3. *revise-op(+A:attribute, +V:value*, +C:constraint, -R:revision)*

   Returns a list of revisions values (or list of values) with associated preferences for an attribute A that violates with value V constraint C. Revisions are not necessarily specific for every value.

4. *preference-op(+R:revision*, -P:preference*)*

   Returns a preference list of the revision. In the original SALT approach this information was acquired from the user.

5. *antagonistic-op(+C:constraint, -S:constraint*)*

   Returns a set of antagonistic constraints where the constraint C is part of. This means that these constraints should be treated as a set. Antagonistic constraints are all re-checked when a revision for one of the elements is applied.

**Domain representation**

Here the graph + cycles etc. should be described.

### D.3.2 Tasks

The top level task of P&R can be considered to build up a theory that specifies a structure of which both the parts and the whole satisfy local constraints. The top level approach of P&R is to construct this theory by selecting small theories for sub-parts or sub-systems and combine these. Following this strategy, the first decomposition is to divide the task into a proposition step, a check step, and a revision step. Propose idetifies initial values to the attributes of the design, check tests the local and global constraints on the design, and revise produces alternative values for violated constraints.

1. *Propose-task(+A:attribute, -V:value)*

   This task takes an attribute and produces a value or a value definition in terms of the values of other attributes. The propose-task can use this value in different ways;

   - If *propos-op* returns a fixed value, then return this value.
   - If *propose-op* returns a definition in terms of other attribute values and the definition is not cyclic (values defined in terms of eachother), then apply the propose task on the attributes of the definition.
   - If the definition is cyclic then ask the user (or another agent) to get an initial value.

2. *check-task(+A:attribute, -OK:boolean)*

   The check task is directly supported by a domain operation and can therefore also be classified as a kads-II PIA.

3. *revise-task(+Attribute, +V:value, R:revision*)*

   Revision operators are supported by the domain model. The difference here is that different revisions are possible and the revision task has to choose a revision from the revision operators in the domain model. The revisions for the various attributes must be chosen in such way that the proposed design converges to a solution, i.e. both local constraint and global constraints are fulfilled.

   Sub-tasks of revise in the P&R approach are:

   (Control: for each of the violated constraints, for each of the values that contribute to the constraint)

   (a) *get-revisions-task(+A:attribute, +C:constraint*, -R:revision*).* This task takes a set of violated constraints and tries to find the possible revisions. In principle any value that contributes to the constrain or the constrained value be proposed in order to be revised.
   (b) *get-preference-task(+R:revision*, -P:preference*)* Get a preference set from the domain knowledge or the user of the different possibilities for revision.

(c) *get_interactions-task(+C:constraint, -I:revision)* Produce a listing of chains of interacting revisions. 'Each chain originates from a constraint whose fixes make other constraints more likely to be violated'.

(d) *select-revision-task(+R:revision*, +IR: revision*, +P:preference, -Fix:revision).* Select a fix to apply based on the possible revisions, the interacting revisions and the preferences.

(e) *apply-revision-task(+/-A:attribute, +R:revision*).* Apply the revision on the attributes of the partial design.

### D.3.3 Inferences

Figure D.3 shows a visualization of an inference structure for the P&R approach.

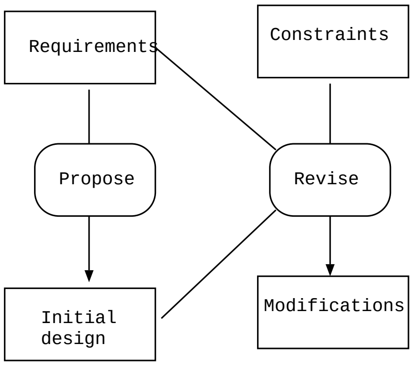



Figure D.3: Inference structure of Propose and Revise

1. *propose(requirements, design)*

   Propose is a binary relation between a list of attributes and a set of values (a partial design). Propose is not reflexive, it is distributive (I Have not checked this yet). We define the propose inference relation as follows:

   There is a binary domain relation P(requirement, value): Propose(R, D) holds when for each element r of R and element d of D S(r, d) holds.

   In design application the relation S refers to 'is satisfied by'.

2. *revise(+A:attribute*, +V:value-set*, +C:constraint*, -R:revision)*

   Differentiate is a relation between a set A (the attributes), a set V (values), a set C (Constraints) and a set R (The design revisions). Differentiate is not reflexive, it is distributive. We define the revise inference relation as follows:

   revise(A, V, C, R) holds if:

   - propose(A, V) holds, the values are propositions for this attribute.
   - constraint(A, R) holds, the revised values satisfy the constraints.

### D.3.4 Dataflow of P&R

The process starts by reading in an initial design with ideal or expected attribute values (see figure D.4). Than the attribute values are evaluated against the requirements and the constraints. If the requirements are not yet met a propose operator is selected and applied. This continues until the requirements have been fulfilled. Than, or in between the proposal phase, the constraints are checked. To revise the design a revision operator is selected. This process continues until a state has been reached were either the the design satisfies the required constraints, and the requirements or no satisfactory design can be found.

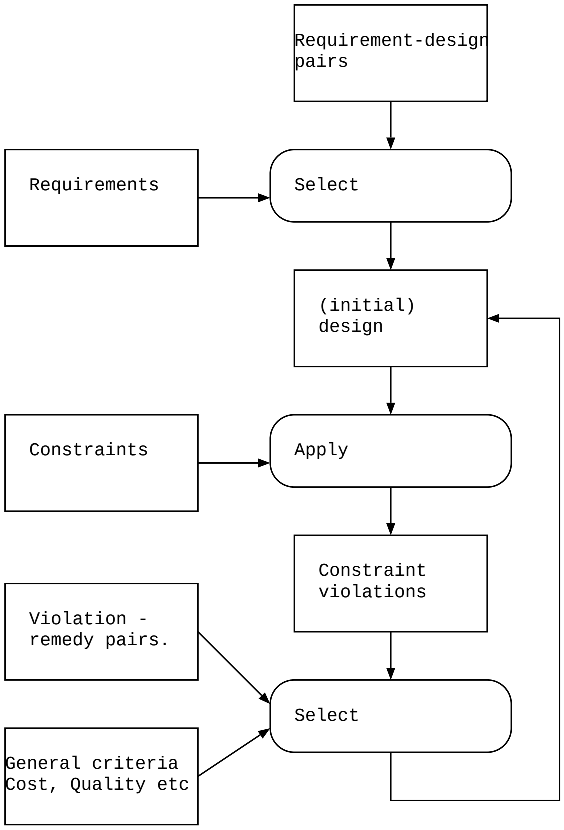



Figure D.4: Data flow of Propose and revise.

### D.3.5 P&R applied on Configuration

Usually the Propose and revise approach is applied on design or configuration tasks. The standard vocabulary of P&R matches therefore directly.

### D.3.6 P&R applied on Scheduling

**P&R assumptions rephrased for scheduling**

- Schedule can be specified by values for a fixed number of attributes.
- Procedures can be specified to determine the starting attributes and their starting values, most likely values and most preferred values' for the schedule.
- There is an enumerable set of schedule constraints.
- For each schedule constraint remedies in terms of modifications to the attribute values of the design can be specified.
- There is not a high level of potential conflict for alternative schedule extensions (original).

### D.3.7 P&R applied on Diagnosis

It is not common to apply the propose and revise RLM on diagnosis. Applying P&R for diagnosis means that we are actually designing a system that has the same behaviour as the malfunctioning device. This seems a possible approach but not very easily applicable due to the domain organization requirements. A second problem is that there are several solutions to design a deviating system using deviating components. When a solution has been found it is not sure whether this actually reflects the state the system under observation is in. Still the approach has some value when the problem is sufficiently constrained, for instance in order to check whether a cause found in another way is in fact able to produce such behavior. But in this case more efficient simulation techniques could also be applied.

#### Task vocabulary

To be done.

#### P&R assumptions rephrased for Diagnosis

- Cause can be specified by values for a fixed number of attributes.
- Procedures can be specified to determine the starting attributes of the cause and their starting values, most likely values and most preferred values.
- There is an enumerable set of cause constraints.
- For each cause design constraint remedies in terms of modifications to the attribute values of the design can be specified.
- There is not a high level of potential conflict for alternative design cause extensions.

## D.4 Extrapolate from a similar case

Here we discuss the original 'extrapolate from a similar case' approach [Marcus, 1988; Offutt, 1988]. The knowledge acquisition system SIZZLE [Offutt, 1988] was created specifically for sizing problems; tasks which have the goal to find the minimum acceptable capacities a system must have, given the use of the system. The 'extrapolate from a similar case' (ESC) approach is not necessarily restricted to the sizing task, but can be used for any task for which the domain assumptions hold. The description we present here will be independent from sizing problems; but we will show how ESC can be applied on sizing tasks and other tasks. We have not extended this description so far that it will cover the complete case based reasoning approaches [Schank et al., 1986; Aamodt, 1991]; because only quantitative extrapolations are handled by the approach described here.
ESC is a RLM in which learning and problem solving are more integrated than in C&D or P&R. Updating the domain models in P&R and C&D is supported by special shells (e.g. MOLE-KA). In ESC the updating of the domain model is another mode of the problem solver; in which the user offers a case together with a solution instead of providing just the case. In the task decomposition and inference structure we have provided we are for the moment only showing an analysis of the problem solving mode of the the ESC approach.

### D.4.1 Domain knowledge

#### Vocabulary

- case: A problem description, solution pair; A given mapping from a point in the source space to a point in the solution space.
- case base: The store of the sources cases.

- source case: An available case in the case base.
- target problem: A problem presented to the problem solver.
- solution adjustment: Modification of the solution description of a source case based on the difference between the problem description of this source case and the problem case.
- solution adjustment model: A specification of how each solution attribute must change with changes in each problem attribute; this model indicates how to extrapolate from a source case.
- problem description: A quantitative description of an input problem.
- solution description: A quantitative description of a solution.
- region updating: Resizing of applicability zones after the introduction of a new source case.
- region: The applicability zone of a case in the problem space.
- similarity metric:
- validated point: Source case.

### Domain Model ontology requirements

- The problem should be described in terms of a mapping from many valued problems to many valued solutions.
- A large set of quantitative source cases is available.
- Similarities among problem descriptions imply similarities among solutions (there are continuous regions).
- No deep explanations necessary.

### Case Models

Here three categories of states are described in whcih the problem solver can be in during problem solving (in contrast with knowledge acquisition).

- In ESC the intermediate case models (states) represent problems, solution, source problem, source solution quadruples. The pairs represent different stages of the case based reasoning process.
- The initial case model (start state) represent the problem with an empty source problem, source solution slots.
- The solution case model (end state) represents a quadruple in which the problem has been matched on a source case, and the source solution has been extrapolated to form the solution.

### Domain operations

The domain model for ESC contains the following types of information:

- Problem, solution pairs; mapping from a point in the source space to a point in the solution space.
- Extrapolation information: pairs of problem attribute differences, solution attribute modifications.
- Applicability zones (regions) of the different cases.

The following operations are directly supported by the domain model:

1. *region-op(Case:case, Region:region).*

   Represents the association between a case and the region of applicability.

2. *source_problem-op(Case:case, Problem:attribute*)*

   Associates a source problem with a case.

3. *source-solution-op(Case:case, Solution:attribute*)*

   Associates a source solution with a case.

4. *adjustment-op(Problem_attribute:attribute, Solution_attribute:attribute*, -Adjustment:procedure)*

   Associates a problem attribute with a (set of) solution attribute(s) and a procedure to adjust the value of source solution to the problem.

**Domain model representation**

Here the domain model representation should be described.

### D.4.2 Tasks

In the problem solving mode[3] the top level task of ESC is equal to the task of C&D, but the theory should be quantitative. The top level approach of ESC is to find this theory by searching through a set of indexed ones. The existing theories are indexed by value regions of a single set of attributes.
Following this strategy, the first decomposition is to divide this task into a matching step and a extrapolation step. Match takes a set of requirements (i.e. problem attributes) and produces an applicable theory (i.e. a source case). Extrapolate takes the theory and changes and applies extrapolations based on the difference between the actual requirements and the source requirements.

1. *match(+Problem:attribute*, -Source_case:case)*

   Returns a source case [4] form the case base that is applicable to Problem. The region of the source problem must cover Problem. Sub tasks are:

   (a) find_region-task(+Target-problem:problem, -Region:region). This task takes a problem and searches through the regions by using the region operator.

   (b) get-source-case(+Region:region, -Source-case:case). Gets an applicable source case from the case base by applying region operator. It is not clear how overlapping regions are can be dealt with.

2. *Extrapolate-task(+Problem:attribute*, +Source_case:case, -Solution:attribute*)*

   Extrapolate takes a problem and a matching case and produces an extrapolation of the source solution based on the difference between the problem and the source problem. Subtasks are:

   (a) *determine-difference(+Problem:problem, +Source-case:case, -Differences:difference*).*

   Takes a problem and a source case, retieves the source problem form the case and compares the attribute values. The differences between the attribute values are returned.

   (b) *find-adjustment(+Difference:difference, -Procedure:procedure)*

   (c) apply-adjustment.

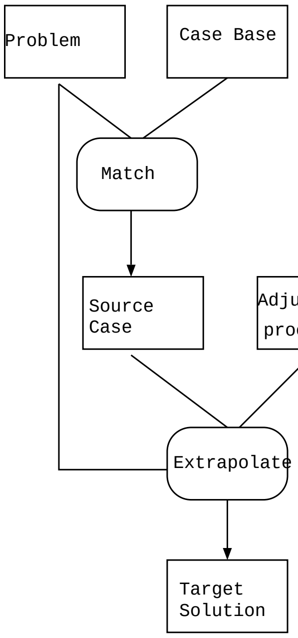



Figure D.5: Inference structure for extrapolate from a similar case at the first level of decomposition.

### D.4.3 Inference structures

An inference structure for ESC at the first task decomposition level is shown in figure D.5.

1. *match* tbd.
2. *extrapolate* tbd.

### D.4.4 Data flow

The complete process starts by reading in the values of the attributes of the problem, and selecting on that basis an applicable region (see figure D.6). From the region the applicable case can be selected. Then a difference computation is made of the source case and the problem. based on this difference the appropiate extrapolation procedures are selected. These are then applied and result into the target solution.

[3]ESC also has a learning mode

[4]In [Offutt, 1988] the matching process is able to find more applicable cases, but overlapping regions seem not to be dealt with; their examples show extrapolations from a single source case per problem.

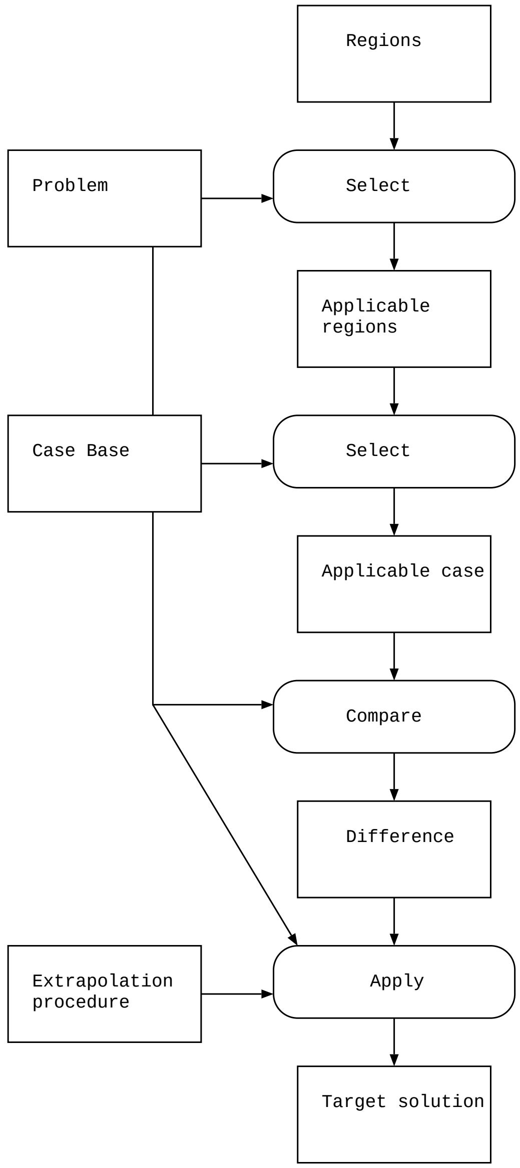



Figure D.6: Dataflow of extrapolate from a similar case

### D.4.5 ESC applied on Sizing

The sizing task consists of finding the minimum accepted capacities a system must have, given the use of the system

| Extrapolate from a Similar Case | Sizing |
|---|---|
| case | sizing case |
| source case | past successful sizing case |
| target problem | a new sizing problem |
| solution adjustment | capacities adjustment |
| solution adjustment model | requirements difference |
| | capacities modification pairs. |
| problem description | amount of different uses. |
| solution description | amount of different capacities. |

**Domain Model ontology requirements rephrased for Sizing problems**

- The sizing problem should be described in terms of a mapping from many valued use of a system to many valued capacities.
- A large set of quantitative source sizing cases is available.
- Similarities among uses of a system imply similarities among minimum capacities of a system.
- No deep explanations necessary why capacities cover the use of the system.

### D.4.6 ESC applied on Diagnosis

The diagnosis task consists of finding an explanation for an event with the minimal cost. The version of ESC presented here is not very applicable on diagnosis because it is based on quantitative regions. Qualitative extrapolation approaches have requirements that fit better on the qualitative nature of diagnosis.

| Extrapolate from a Similar Case | Diagnosis |
|---|---|
| case | Symptoms, explanation pair |
| source case | Past successful symptom, explanation pair |
| target problem | A new complaint |
| solution adjustment | Adjustment of explanation |
| solution adjustment model | symptoms difference |
| | explanation modification pairs. |
| problem description | Complaint |
| solution description | Event that explains complaints. |

**Domain Model ontology requirements rephrased for Diagnosis**

- The Diagnosis problem should be described in terms of a mapping from many valued symptoms of a system to many valued explanations.
- A large set of quantitative source symptom, explanations pairs is available.
- Similarities among symptoms of a problem imply similarities among explanations of the problem.
- No deep explanations necessary why explanations cover the symptoms.

### D.4.7 ESC applied on Classification

The problem for classification is similar to diagnosis; most classification problems are best described qualitatively.

| Extrapolate from a Similar Case | Classification |
|---|---|
| case | Features, class pair |
| source case | Past successful features, class pair |
| target problem | A new instance |
| solution adjustment | Adjustment of matching class |
| solution adjustment model | features difference |
| | class modification pairs. |
| problem description | set of features |
| solution description | Class that matches on instance. |

**Domain Model ontology requirements rephrased for Classification**

- The Classification problem should be described in terms of a mapping from many valued symptoms of a system to many valued classes.
- A large set of quantitative source feature, class pairs is available.
- Similarities among features of an instance imply similarities among the matching class of the instance.
- No deep explanations necessary why classes match on the instance.

### D.4.8 ESC applied on Assessment

| Extrapolate from a Similar Case | Assessment |
|---|---|
| case | case features, decision pair |
| source case | Past successful case features, decision pairs |
| target problem | A new assessment case |
| solution adjustment | Adjustment of matching decision |
| solution adjustment model | case features difference |
| | modification pairs. |
| problem description | set of case features |
| solution description | Decision that matches on assessment case. |

**Domain Model ontology requirements rephrased for Assessment**

- The Assessment problem should be described in terms of a mapping from many valued symptoms of a system to many valued classes.
- A large set of quantitative source case feature, decision pairs is available.
- Similarities among case features imply similarities among decisions.
- No deep explanations necessary why case features lead to a assessment decision.

### D.4.9 ESC applied on Design

| Extrapolate from a Similar Case | Design |
|---|---|
| case | requirements, design pair |
| source case | Past successful requirement, design pairs |
| target problem | A new set of requirements |
| solution adjustment | Adjustment of matching design |
| solution adjustment model | requirements |
| | design modification pairs. |
| problem description | set of requirements |
| solution description | Design that fulfills requirements. |

**Domain Model ontology requirements rephrased for Design**

- The Design problem should be described in terms of a mapping from many valued symptoms of a system to many valued classes.
- A large set of qualitative source requirements, design pairs is available.
- Similarities among requirements imply similarities among design.
- No deep explanations necessary why requirements lead to a design.

## D.5 Comparison

In this section we have presented descriptions of some RLM's. Now it is interesting to see where the real differences are between these approaches are.
C&D and P&R are different RLM's because different domain organizations have been chosen for them. If we consider both RLM's to build up a theory of their problem, then we see that they have different approaches in building up that theory. In P&R one theory is proposed (the initial design) and this theory is stepwise refined by propose and revision operators. In C&D several small theories are available which are selected and tested until one single theory remains. Therefore C&D requires the availability of applicable theories indexed by features, while P&R requires the availability of theory modification operators and theory evaluation mechanisms.
Comparison of the rest: to be done.

## D.6 Modelling Problems

1. It is a problem what pieces of knowledge wil be classified as inference relations and functions. In the C&D approach cover maps easilyt to the inference relation idea, but differentiate is too complex; it is a task. Makes it sense to represent tasks whithin an inference structure?
2. In the C&D apporach there exist cover inference relations, and relationsbetween events like enabling, required etc. Are the relations inference relations? I woudl say not, they are not inferred, so call them domain relations. But then also cover will be a domain relation and not an inference.

# Appendix E

# Library Components - Methods and Tasks from CONSTRUCT

**Agnar Aamodt**

VUB AI-Lab

This appendix suggests a few generic models, more specifically some problem solving methods for construction type tasks. The material provided here is interesting as it applies the idea of generic model as we have proposed it in Chapter 5. Apart from this it gives extensive lists of methods involved in construction tasks. These have not described in equal detail. The examples that are provided are completed with an extensive discussion on the intended interpretation, to illustrate that having a well-defined BNF in only one part of the problem. The current CML definition defines a basic concept ontology, a set of basic relations between concepts, and a structure in which to express concpetual models, but it does not specify in detail how to actually describe various types of knowledge contents - i.e. how to write down the contents of the slots of model frames. For one of the examples described in this appendix, two different ways to do this - a declarative and a functional one - are illustrated.
The material on models and methods in this appendix is derived from results of the CONSTRUCT project.

## E.1 Introduction

This appendix presents several types of problem solving methods and tasks, as identified within the Construct project. They are partly very general entities, as described in the general methodology report [Steels, 1991a], partly entities derived from more specific studies and application descriptions. The methods and task types studied in Construct are those that may be useful for construction type applications (design, planning, configuration, scheduling, etc.). This does not exclude some of these entities from also being useful - at some level of generality - for solving other types of problems.
The next section describes methods, the subsequent one describes tasks. The main focus is on methods, at present, task types are presented merely as a list, without accompanied example descriptions. Each section is divided into an enumeration of all entities identified up till now,

followed by a description of some entities assumed to be relevant for the CommonKads library. The description language used is based on the current versions of the D1.1. and D1.3.
Both sections first list a set of problem solving methods, classified as done in Construct. This is followed by descriptions of example library components. Example descriptions of three problem solving methods and one task type are given, as possible model frames definitions. Given the current state of the model description language and library structure, these examples should not be regarded as illustrations of a final, established description language, but rather as input to the continued work on refining this language (i.e. the CML) and structuring of the library.
The purpose of this appendix is therefore twofold: First, to provide a list of candidate components for the library, suitable for modeling construction type applications. And second, to provide examples of rather complete component descriptions, within the framework of the current D1.1 version. This should give feedback to the questions of what type of components we want in the library - i.e. their granularity size and complexity ('combinedness'), as well as how to actually use the CML language to express the structure and contents of library components.

## E.2 The COMMET Framework

The methods and task types described in this appendix will use the terminology of the COMMET framework when classifying and listing the candidate components, but the example descriptions will follow the CommonKads terminology, from D1.1 - or, rather, my interpretation of it, since some of the terms still need a clarification.
In order to get a rough impression of the similarities and differences between the COMMET and CommonKads methodologies - at a high level of abstraction at least - main features of COMMET are summarized in this section.
A knowledge level description of an intelligent system, according to COMMET, is made by splitting the analysis and/or design work in three, corresponding to three *perspectives* of a knowledge-level description. These perspectives are *tasks*, *models*, and *methods*.

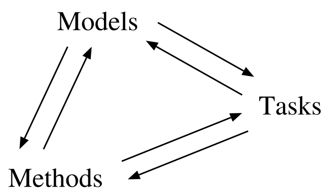



Figure E.1: The three perspectives

Models, here, refer to domain models of the application domain, tasks are tasks/subtasks identified in order to solve an application problem, and methods are problem solving methods for decomposing tasks into subtasks, or for solving a task without further decomposition.
Models and tasks are structured into task decompositions and model dependency graphs, respectively. This is illustrated below.
Model dependency diagrams are used to relate the various types of domain knowledge which is needed to construct a new case-model on the basis of existing case models. The illustration in figure E.2 illustrates that an initial case model[1] is constructed from user input and input from a particular model of domain knowledge (e.g. a model defining a set of relevant problem features). This case model - called case-model-1 in the figure - is in turn input to another construction activity, which takes another domain model (e.g. a causal model, or a combined structural and associational model), and constructs a second case model.

[1]The notion of case model is roughly the same as in CommonKads

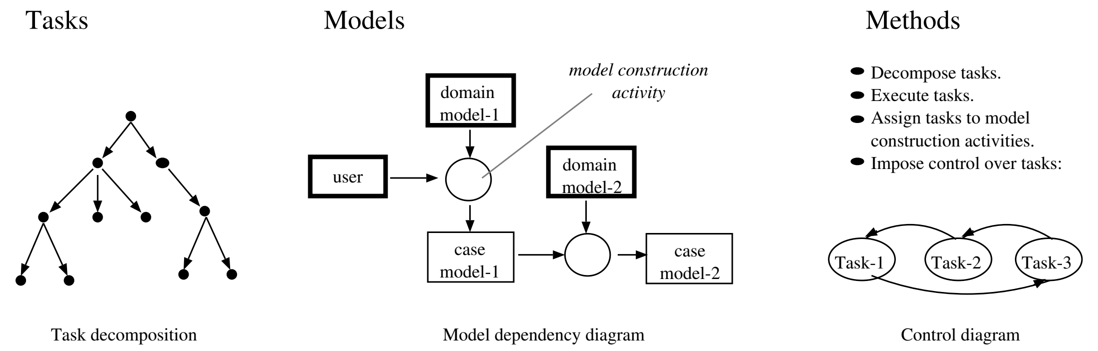



Figure E.2: Structures and roles within the three perspectives

The power of using the three perspectives for knowledge level modeling lies in how the perspectives interact. When making a description according to one perspective, the other two perspectives - or one of them - may be used to make the description more detailed and application focused.
As an example of interacting perspectives, let us take a task perspective, and see how a task may be decomposed. A task may be decomposed in two principled ways: By a method-oriented decomposition, or by a domain-model-oriented decomposition. In the latter, the subtasks of a task is determined by what type of domain-models they relate to and the type of case-models they produce, regardless of the problem solving method used to achieve the decomposition. In CommonKads, the first view is adopted. A model-oriented decomposition is somewhat analogous to an inference structure in CommonKads - if tasks are mapped to inferences - since it relates tasks to domain model entities. A difference which still remains, however, is that inference structures in CommonKads relate inferences to knowledge *roles* only, not to the particular domain models (i.e the actual model components) involved.
It should be noted that the Construct project as well as the Commet framework is still under development. The descriptions given here are based on the currently available documents (March 1992).

## E.3 Problem Solving Methods

### E.3.1 List of Methods

Below, a set of general methods are listed. A this level of generality, some of these methods also apply to non-construction tasks. More specialized methods for construction tasks have been identified for some of the applications described in the Construct project. Examples of such methods are given in chapter E.3.3.2, as part of the description of a combined, specialized method.

- **Task decomposition methods**
  Each of these methods is associated with a task and its decomposition into subtasks, as well as a control structure over these subtasks.

  – *Divide-and-Conquer*
    A task is decomposed into subtasks which may further be handled independently. The result from each subtask is then used to construct a task solution. The control is sequential.

  – *Progressive-Refinement* A task is decomposed into a set of gradually more specific task descriptions. An abstract description of the task is first made. After a solution of this task is presented, more details are added to the task description. This, in turn, leads to a more refined solution, and so on. Control is basically sequential, but iterative loops back to more abstract descriptions may be necessary.

This method is different from Divide-and-Conquer. It does not decompose a task into independent subtasks from which the final solution is constructed. Instead, a single, complete solution is worked on all the time. The subtasks are not partial tasks, but a sequence of more specialized descriptions of the task.

  * *Hierarchical-Design*
  * *Hierarchical-Planning*
    Progressive refinement applied to design and planning tasks, respectively.

- *Propose-and-Revise*
  A task is decomposed into three subtasks corresponding to proposing an initial solution, assessing the proposal, and revising the solution based on the assessment. The control is basically sequential, but may contain iterative loops, since a proposed solution may turn out to be too 'wild' to be revisable.

  * *Case-Based Reasoning*
    A solution is proposed by matching the task description against a set of previously solved cases, i.e. collections of specific domain knowledge, and adapting the solution of the best matching case to the current task. Domain models describing general domain knowledge are typically used to assess case similarity, derive and check consequences of proposed solutions, and to modify past solutions. (Note that it is the reasoning part of the term 'case-based reasoning' that is focused here, since the term, as normally used, incorporates a learning method as well.)

• **Task execution methods**
Methods that execute bottom level tasks - leaf tasks - in a task decomposition.

- Mapping methods
  Methods that use one or a set of case models to build or modify another (set of) case models, e.g. from a case model of the problem description to a case model of the solution.

  * *Linear-Mapping*
    A method that maps many input descriptors of a task to one output descriptor.
  * *Differentiation*
    Method that identifies the most discriminating hypothesis on a differential, and uses this to look for supporting evidence.
  * *Top-Down-Refinement*
    Uses a domain class tree to discriminate downwards, step-by-step.
  * *Weighted-Evidence-Combination*
    Weighing relative importance of evidence items for an hypothesis.
  * *Similarity-Matching*
    Uses a metric to compute similarity between source case and target case.

- Expansion methods
  Methods that transform the contents of a single case model

  * *Constraint-Propagation*
    A method ('expansion method') that modifies a case model by propagating constraints defined in a domain model.

• **Search methods**
The mapping and expansion methods aims at providing a unique way to achieve a mapping or an expansion, respectively. A mapping method, for example, identifies the necessary domain

models so that a set of input case model will map to a single output case model. However, the domain model may not be strong enough to map into a single output case model, but rather point to several possible case models. This corresponds to several optional paths in a space of case models. Hence, search is needed to decide along which path to proceed. The search process will explore the alternative choices, and select the most promising one - according to some criteria.

Three well-known types of search methods are:

- *Depth-First-Search*
- *Breadth-First-Search*
- *Heuristic-Search*

A range of search methods and their algorithms have been identified and described within Construct. Most of these methods are very specific, and primarily of interest for symbol level realizations. As an illustration, parts of the search method set described in the project is listed below.

- Admissible Algorithms:
  - *A**
  - *IDA**
- Efficient Algorithms:
  - *A(epsilon)**
  - *Beam-Search*
- Fast convergence algorithms:
  - *SA**
  - *RTA**

Task decomposition methods, task expansion methods, and search methods are the three main types of methods in Commet. As a result of analyzing and describing a range of applications within Construct, using the Commet framework, several *combined methods* have been implicitly identified. For example, in an application describing design of kitchens, the resulting task structure was achieved by applying different decomposition methods for different tasks in the structure. The usefulness of looking into these combined, or instantiated, problem solving methods, is that they may serve as candidates for a set of specialized generic components in our CommonKads library. A method for solving kitchen design problems is probably too specific, but if it is generalized into a method for configuring object in a room, it may be at a usable (re-usable) level.

### E.3.2 Combined methods

Methods that are somewhat specific in the sense that they also specify task knowledge and domain knowledge related to the method.

Combined methods typically result from previous model development efforts. A problem solving method in which domain knowledge as well as task decompositions is included, describing a particular application, is definitely a combined method, but it will usually be too specific for later reuse. But such an application-specific method may be somewhat generalized in order to apply for certain *types* of application problems. In section E.3.3.2 such a generalization has been made, based on an example application of kitchen design [?], giving a combined method called:

- *Hierarchical-Configuration-of-Room-Objects*

### E.3.3 Descriptions of Problem Solving Methods

In this section, candidates for the CommonKads library are described. The description is casted in an *expertise model frame* (or knowledge model frame) as described in the main part of this document. That is, each description corresponds to the filling of the 'conceptual description slot' in the top level frame structure - which means a description in terms of the CML language. Within this dimension, the object descriptions are grouped according to the four knowledge type entries of that frame structure: methods, tasks, domain, and inference.
The CML, as suggested in the D1.1 Appendix, proposes an *object structure*, i.e. a set of descriptors (slots), for describing CommonKads objects - including library components. The descriptions below are kept as close to this CML structure as possible. Although an initial suggestion for a language to describe the *contents* of these slots also has been presented - in the form of examples in D1.1 - no decisions has been taken regarding that part of the CML. In order to illustrate two different types of such languages, the first example below, the Divide-and-Conquer method, is described twice, using a functional and a declarative style language, respectively. The reason for including both languages is, partly, that the actual description was made in these two steps, partly that comparisons of expressivity and user transparency of the two languages then can be made.
Each library component description below is accompanied by a textual explanation.

#### E.3.3.1 A Very General Method

**Method definition in a procedural style:**

Procedure names and other predefined terms are written in uppercase. The meaning of these must be clearly defined; most of all to the user, but also to the system - to the degree that some system support (in terms of operations over the procedural structures) will be provided. They could, for example, be defined as a set of functions, or predicates, that would lead to prompting of the user, checking of consistency, generation of consequences, or some other action appropriate for a given procedure term.

```
DIVIDE-AND-CONQUER
 knowledge-type      task-decomposition-method
 method-knowledge
   goal               [FIND-A Task-solution BY
                        DECOMPOSE Task INTO INDEPENDENT Sub-Tasks
                        SOLVE EACH Sub-task
                        COMPOSE Solution FROM ALL Sub-task-solutions]
   ps-methods-used    decompose-into-independent-subtasks,
                      compose-solution-from-subtask-solutions
   method-definition  [GIVEN-TASK Task
                        USE-METHOD decompose-into-independent-subtasks TO
                         DECOMPOSE Task INTO Task-decomposition SUCH-THAT
                           ALL Sub-tasks IN Task-decomposition
                             ARE SUFFICIENTLY INDEPENDENT
                         FOR-EACH Sub-task IN Task-decomposition
                          USE-METHOD a problem-solving-method TO
                           SOLVE Sub-task
                        USE-METHOD compose-solution-from-subtask-solutions TO
                         COMPOSE Task-solution FROM
                           ALL Sub-task-solutions OF Sub-tasks
                             IN Task-decomposition]
 task knowledge
   task-characterisation
                      [POSSIBLE-TO FIND-A Task-decomposition OF Task SUCH-THAT
                         ALL Tasks IN Task-decomposition
                          ARE SUFFICIENTLY INDEPENDENT
                        AND
```

```
                         POSSIBLE-TO FIND-A Task-solution OF EACH Sub-task]
domain-knowledge
  domain-model-assumption
  case-model-assumption
                      [Case-model HAS-DESCRIPTOR FOR
                         Input-case-specification-of-Task
                         Input-case-specification-of-Sub-Task
                         Output-case-specification-of-Task
                         Output-case-specification-of-Sub-Task]
inference knowledge
  inferences-used     generate-independent-subcomponents IN DECOMPOSE,
                      check-mutual-independence IN DECOMPOSE,
                      compose-solution-from-partial-solutions IN COMPOSE
  inference-structure generate-independent-subcomponent(Input-case-
                        specification-of-Task, Input-case-specification-
                        of-Sub-Task)
                      check-mutual-independency(Input-case-specification-
                        of-Sub-Task, Modified-input-case-specification-
                        of-Sub-Task)
                      compose-solution-from-partial-solutions(Output-case-
                        specification-of-Sub-Task, Output-case-specification
                        of-Task
```

The intended interpretation of the description is as follows:

The *knowledge-type* of Divide-and-Conquer is a task decomposition method. It decomposes a task into subtasks without saying anything about how the subtasks may be solved. This is method type is assumed to be named in the typology of problem solving methods, which in turn is a type of *generic model*.
The *goal*, as described here is a Lotte more specific than goals were specified in D1.1. The idea is that the goal must say more than just 'Find a solution', since this general goal should be stated in the top level object of the typology, i.e. the 'problem solving method'. It is assumed that, normally, descriptions can be inherited along the typology links, so that this most general goal would automatically also be a goal for the Divide-and-Conquer method, unless a more specific goal is specified. Here, a constraint on how the solution is to be found, i.e. an abstraction of the underlying method, is pulled into the goal, as an example of a more specific goal description. It expresses that the goal of this particular method is not only to find a solution, but to do this by finding a decomposition into independent subtasks, and using their solutions to compose a task solution.
The *ps-methods-used* slot lists the sub-methods of a method being described. Here, two partial problem solving methods are used within the Divide-and-Conquer method. These terms are also identifiable as problem solving methods, by being defined within the method typology referred to above.
The *method-definition* slot is where the actual method, including its control, is described. In this case the control is sequential. We may want to explicitly describe this property (sequential control) in a separate slot ('control type') or something, but that is not done done in this example. The method description incorporates the two partial methods named in the ps-methods-used slot, and relates them to what they do, described in the functional description language. In addition, a non-specified method - 'a problem solving method' - is referred to for solving the subtasks.
The *task-characterization* defines the properties of tasks for which this method applies. It describes one or more conditions on the task. As mentioned above, all uppercase terms refer to functions which should have a clear semantical interpretation. For example, 'ARE SUFFICIENTLY INDEPENDENT' has to be given a meaning by specifying a set of criteria. These criteria are likely

to be domain dependent, in which case hooks to the domain category should be provided. The necessary domain knowledge description can then be filled in during instantiation of this method in a particular modeling process.
The *domain-model-assumption* and case-model-assumption expresses dependencies of the method of particular properties of domain knowledge. The case model must provide for descriptors that enable a coupling to the dynamic knowledge roles referred to by the inferences. There are also domain model assumptions, given the fact that whether, and in what way, a task is decomposable into sufficiently independent subtasks is influenced by properties of the domain knowledge. This is not easy to express in a structured way - but at least a verbal description should be given.
The *inferences-used* and *inference-structure* descriptors list the inferences which are used by the method. Inference in this sense is not a notion in COMMET, so the list exemplifies a set of inferences described according to the D1.1 framework. The inference structure, in addition, also gives the dependencies (the links) to knowledge roles. Here, only the dynamic roles are listed. There are also dependencies, and assumptions, about static roles, since the way a task may be decomposed depends on some characteristics of the domain knowledge.
**Method definition in a declarative style:**

As above, predefined terms are written in uppercase. The meaning of these must be clearly defined, e.g. by revering to standard first-order predicate logic.

```
DIVIDE-AND-CONQUER
 knowledge-type      task-decomposition-method
 method-knowledge
  goal                G: FIND(S:Solution)
  ps-methods-used     DECOMPOSE: decompose-into-independent-subtasks,
                      COMPOSE: compose-solution-from-subtask-solutions,
                      SOLVE-TASK: a problem-solving-method
  method-definition   A1: FORALL x,y (DECOMPOSE(x,y) IMPLIES Task-decomposition(y,x))
                      A2: FORALL x,y,z
                           (EXISTS w ((Solution(x,y) AND Task-decomposition(z,y))
                                      IMPLIES
                                      COMPOSE(x, Solution(w,z))))
                      A3: FORALL x,z
                           (EXISTS y,w ((DECOMPOSE(x,y) AND SOLVE-TASK(y,w)
                                         AND COMPOSE(z,w))
                                        IMPLIES Solution(z,x)))
 task-knowledge
  task-characterisation
                      TC1: Task(x:Task) AND Sub-task(z:Sub-task,x)
                           AND Task-decomposition(y:Set-of(z),x)
                      TC2: FORALL x
                            (EXISTS y
                             (Task-decomposition(y,x)
                                AND
                              FORALL w,z (Pair-of-elements((w, z), y)
                                          AND Sufficiently-independent(w, z))))
                      TC3: FORALL x
                            (EXISTS y,w
                             (Task-decomposition(y,x))
                               AND
                              FORALL z ((Sub-task(z,y) IMPLIES Solution(w,z))))
 domain-knowledge
  domain-model-assumption
  case-model-assumption
 inference-knowledge
```

```
inferences-used     generate-independent-subcomponents IN DECOMPOSE,
                    check-mutual-independency IN DECOMPOSE,
                    compose-solution-from-partial-solutions IN COMPOSE
inference-structure generate-independent-subcomponent(Input-case-
                      specification-of-Task, Input-case-specification-
                      of-Sub-Task)
                    check-mutual-independency(Input-case-specification-
                      of-Sub-Task, Modified-input-case-specification-
                      of-Sub-Task)
                    compose-solution-from-partial-solutions(Output-case-
                      specification-of-Sub-Task, Output-case-specification
                      of-Task
control             DECOMPOSE(Task, Set-of(Sub-task)),
                    FOR-ALL Subtask IN Set-of(Sub-task)
                      SOLVE-TASK(Sub-task, Solution(Sub-task))
                     RESULTING-IN
                      Set-of(Sub-task-solution),
                    COMPOSE(Solution, Set-of(Sub-task-solution)).
```

The above description differs from the previous in that a specific goal is not ascribed, so the goal is the general one valid for all problem solving methods, and that the control structure of the method is explicitly described in a separate slot.
Graphical illustration of the inference structure for this method is shown in figure E.3. The inference structure contains two parts, tied together by the problem solving method for solving the subtasks. It seems that we need a notation for incorporating problem solving methods into inference structures, unless we only want to talk about inference structures for leaf tasks, i.e. tasks which are solved entirely within the task itself.

### E.3.3.2 A Rather Specific Method

In this section, a more specific, combined method is described. Separate methods are contained within method structures of combined methods, which include several levels of task decomposition methods as well as task execution methods. The methods described in the source example is somewhat generalized in the following description, in the sense that very application-specific terms (e.g. 'kitchen') are replaced by more generic terms (e.g. 'room').
The combined method (combined model?) described is called 'Hierarchical-Configuration-of-Room-Objects', based on the kitchen-design example [Geldof, 1991]. In the cited report, the top level task, kitchen-design, is decomposed into two tasks: room-space-allocation and kitchen-configuration. For the combined method described here, we start with the kitchen-configuration task as our top task - which we rename: room-configuration.
The decomposition method used for this task is a specialization of *hierarchical-design*. The resulting tasks are called abstract-design and detailed-selection.
(Note that the problem solving methods referred to in this description are most often specializations of more generic methods, as in the previous paragraph. It is an open question how specific methods we want to keep in the library. In the next paragraph an example of a more specific method is given.)
The method of the abstract-design task is a two-step process of first identifying the intervals of possibly useful floor space, followed by an assignment of classes of objects to the available intervals. This method will here be referred to as the *identify-space-and-assign-objects* method.
The method of the detailed-selection task is *divide-and-conquer*, splitting the task into local configurations within subareas, followed by a global configuration of the entire kitchen. The problem solving methods for the first of these leaf tasks is a method ('expansion method') we will refer to as *eliminate-by-propagation-of-constraints*. Unlike the task decomposition methods previously

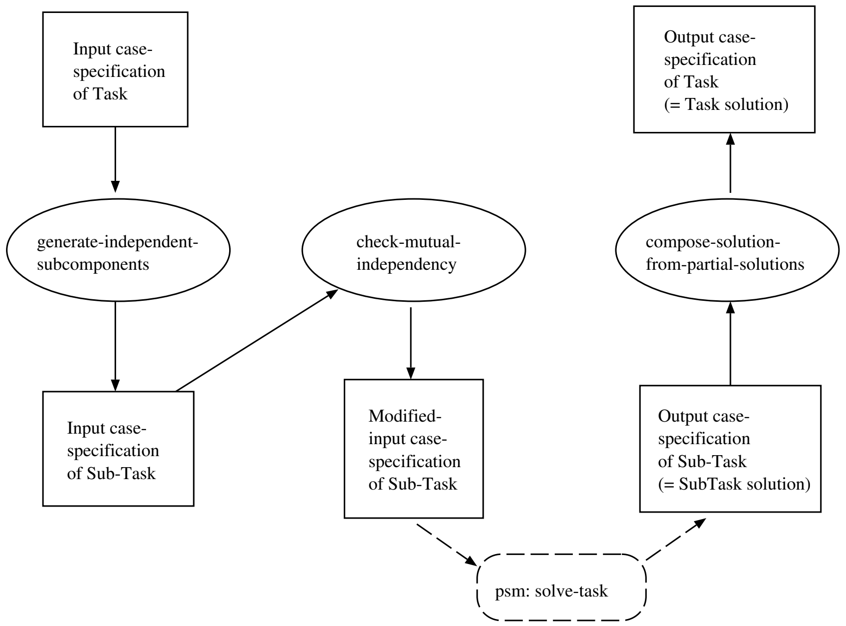



Figure E.3: Inference structure for the generic Divide-and-Conquer method. Note that an additional structure (dashed arrows) is introduced in the lower right part of the figure, in order to link the two separate inference structure related to inferences before and after the application of the subtask solving PSM. All arrows can be viewed as input/output flow of information related to the problem case.

mentioned, which all have a sequential control structure, this methods iterates between determination of constraints, generating possible configurations, and eliminating unsatisfactory ones, until all constraints have been satisfied. For the second leaf task (global configuration), a version of the *generate-and-test* method is used. Since they are bottom-level methods, these methods will have their inferences specified and interlinked in an inference structure (not specified in the source example).

```
HIERARCHICALLY-CONFIGURE-ROOM-OBJECTS
 knowledge-type      combined-problem-solving-method
 method-knowledge
  goal                [FIND-A Configuration
                        OF Physical-Objects IN Room
                        SUCH-THAT
                          Functional-Constraints AND Physical-Constraints
                        ARE SATISFIED]
  ps-methods-used     design-hierarchically, identify-space-and-assign-objects,
                      find-areas, associate-object-classes-to-areas,
                      divide-and-conquer, eliminate-by-propagation-
                      of-constraints, generate-and-test, generate-
                      configuration, test-configuration
  method-definition   [GIVEN-TASK room-configuration
                         USE-METHOD design-hierarchically TO
                           DECOMPOSE room-configuration INTO
                             abstract-design AND detailed-selection
                         USE-METHOD  identify-space-and-assign-objects TO
                           DECOMPOSE abstract-design INTO
```

```
                              finding-areas AND
                              associating-object-classes-to-room-areas
                          USE-METHOD  find-areas TO
                            EXECUTE finding-room-areas
                          USE-METHOD  associate-object-classes-to-areas TO
                            EXECUTE associating-object-classes-to-room-areas
                          USE-METHOD divide-and-conquer TO
                            DECOMPOSE detailed-selection INTO
                              finding-satisfactory-local-layout AND
                              finding-satisfactory-global-layout
                          USE-METHOD eliminate-by-propagation-of-constraints TO
                            EXECUTE finding-satisfactory-local-layout
                          USE-METHOD generate-and-test TO
                            DECOMPOSE finding-satisfactory-global-layout INTO
                              generating-possible-layouts AND
                              testing-layouts
                          USE-METHOD generate-configuration TO
                            EXECUTE generating-possible-layouts
                          USE-METHOD test-configuration TO
                            EXECUTE testing-layouts]
  task-characterisation
                       [THERE-EXISTS-A DESCRIPTION-OF Room, Physical-Objects,
                          Functional-Constraints, Physical-Constraints
                         AND
                          POSSIBLE-TO SATISFY ALL Constraints
                          BY HIERARCICAL Task-decomposition]
  tasks-involved       room-configuration, abstract-design, detailed-
                       selection, finding-room-areas, associating-object-
                       classes-to-room-areas, finding-satisfactory-local-layout,
                       finding-satisfactory-global-layout, generating-
                       possible-layouts, testing-layouts
  task-structure       (room-configuration
                        (abstract design
                         (finding-room-areas)
                         (associating-object-classes-to-room-areas))
                        (detailed-selection
                         (finding-satisfactory-local-layout)
                         (finding-satisfactory-global-layout
                          (generating-possible-layouts)
                          (testing-layouts))))
 domain-knowledge
  domain-model-assumptions
                       room-with-configurable-areas, configurable-objects,
                       user-preferences,  object-area-associations,
                       configuration-layout
  domain-structure     physical-functional-dependencies-of-objects-and-groups,
                       dependencies-within-objects-groups-and-areas
```

The object description differs from the previous Divide-and-Conquer method in that the task knowledge is included, domain knowledge structures are included, and the inference knowledge is empty. The reason for the latter is that the inference knowledge is assumed to be described as part of each separate method that make up this combined method. This method incorporates a particular task structure, and a particular set of (types of) domain models, which therefore are explicitly named.

## E.4 A list of Task types

In Construct, a set of task types have been identified. No elaborate descriptions of these have yet been made with respect to CommonKads. As an initialization of this work, a list of task types so far identified within Construct is given.

- **Task Type***Design*
  Atomic Tasks:
  - *Specifications-Formulation*
  - *Sites-Allocation*
  - *Component-Placement*
  - *Component-Selection*
  - *Component-Parameterization*
  - *Component-Transformation*
  - *Components-Composition*
  - *Logical-Configuration*
  - *Connections-Design*
  - *Functional-Simulation*
  - *Simulation-of-the-Physical-Behavior*
  - *Architecture-Design*

  Compound Tasks:
  - *VLSI-Design*
    Specifications-Formulation + Architecture-Design + Component-Selection + Functional-Simulation + Sites-Allocation + Component-Placement + Connections-Design
  - *Computer-Configuration*
    Component-Selection + Component-Parameterization
  - *Mechanical-Configuration*
    Component-Selection + Component-Parameterization + Component-Placement + Simulation-of-the-Physical-Behavior
  - *Spatial-Configuration*
    Component-Selection + Component-Placement
  - *Form-Features-Design*
    Component-Selection + Components-Composition + Component-Transformation
- **Task Type***Assignment*
  Compound Tasks:
  - *Stores-Localization*
  - *Office-Planning*
  - *Traveling-Salesman*
  - *Car-sequencing*
- **Task Type***Scheduling*
  Atomic Tasks:

- *Manpower-Assignment-to-Activities*
- *Manpower-Assignment-to-Time-Slots*
- *Manpower-Assignment-to-Locations*
- *Equipment-Assignment-to-Activities*
- *Equipment-Assignment-to-Time-Slots*
- *Equipment-Assignment-to-Locations*
- *Activities-Assignment-to-Time-Slots*
- *Schedule-Repair*
- *Activities-Generation*
- *Activities-Regroupment*

Compound Tasks:

- *Job-Shop-Scheduling*
  Equipment-Assignment-to-Activities + Activities-Assignment-to-Time-Slots + Schedule-Repair
- *(Airline or Railway) Staff-Scheduling*
  Manpower-Assignment-to-Activities + Manpower-Assignment-to-Time-Slots + Manpower-Assignment-to-Locations + Activities-Generation + Activities-Regroupment
- *(Airline or Railway) Equipment-Scheduling*
  Equipment-Assignment-to-Activities + Equipment-Assignment-to-Time-Slots + Equipment-Assignment-to-Locations + Activities-Generation + Activities-Regroupment
- *(College) Timetable-Scheduling*
  Manpower-Assignment-to-Time-Slots + Activities-Regroupment + Manpower-Assignment-to-Activities

# Appendix F

# Technical Diagnosis by Top-Down Fault-Localization

**Klas Orsvärn**

SICS

This application analysis describes how a generic model for technical diagnosis can be refined and modified to support the features of a specific application, which is a KBS that guides a service technician in the task of diagnosing a specific measuring instrument. The initial generic model is based on the interpretation model of Systematic Diagnosis By Localization. There are two features of the application th+at Systematic Diagnosis does not provide support for: the number of observations must be kept down, and a part cannot be tested independently of other parts. A number of modifications and refinements of the initial generic model are described. They are based on an analysis of how the features are handled in this specific application, but they should be more generally applicable. The result of this work could form a basis for a generic model for technical diagnosis. It could serve as a detailed case-study of how to create a conceptual model by using a library of generic modelling components to satisfy the features of an application.

## F.1 Relation to the library framework

The work reported here has been in progress until recently. Therefore, no attempt has yet been made to adhere to the details of CML. The description is focused on task decomposition and control. Other aspects are verbally described. At the lowest level of decomposition, which corresponds to primtive infrences, the required domain knowledge is verbally described. The hypothesis is that other parts of the conceptual model can be derived from the existing description, e.g. domain layer and inference structure.

No attempt has been made to stick to any typology of basic modelling components, e.g. primitive inferences. Once we decide on such a typology, the models will have to be restructured or refined.

## F.2 Introduction

The application that is the basis of this analysis is a KBS that guides a service technician in the task of diagnosing a specific instrument for testing telecommunications equipment. It is one of the applications under the header Alda in [Orsvärn, 91a] and [Orsvärn, 91b]. The KBS was developed by Infologics and Swedish Telecom in 1989 – 1990. The application is henceforth called HP. It is a consultation system that makes a sequence of requests for tests to be performed by the user, who inputs the results of each test to the system. The system finally reports which part of the device should be replaced. The CML components described here are the results of reverse engineering. However, the intention is that the presentation should reflect a possible development process of this application.

The existing KBS is specified as a decision tree (with some extensions) and implemented as a rule-based system. What I have done is to explicate assumptions that the tree is based on. The assumptions have been derived by trying to justify the decisions in the tree, by studying technical documentation of the HP device, and by interviewing the knowledge engineer and domain expert who developed the system.[1] Those assumptions form the basis of a the generic modelling components presented here. The next step in this work will be to describe the system as an application of that method. The analysis can be described as a process of reverse engineering. In this process I found a number of weaknesses in the system, that essentially correspond to false assumptions. The reason why they have not been discovered before is partly because they are left tacit in the original specification and implementation, and partly because the system has not been extensively used. This made it more difficult to extract generic modelling components. It would have been easier to create an analysis which is closer to the implementation, but the result would not have been reusable. Nevertheless, I have found a number of useful strategies. The major weakness in the system is that the strategies are sometimes applied incorrectly and inconsistently. The analysis presented here are only based on strategies that I have found in HP. I have not yet reached the stage of re-specifying the system as an application of this set of modelling components, but I plan to do that in the near future. That will of course be necessary to validate the result.

As a starting point I use the interpretation model of Systematic Diagnosis by Localization, as it is described in [Breuker et al, 1987] and [Wielinga et al, 1992], since it captures a central strategy used in HP. To avoid confusion with the original description of Systematic Diagnosis, I will use the name Top-Down Fault-Localization (TDFL) for the set of CML components presented here.

I will begin with an initial version which is my reinterpretation of Systematic Diagnosis by Localization. The original presentation of Systematic Diagnosis is intended to be domain independent. However, my interpretation is dependent on particular kinds of domain models. I will then discuss the assumptions on which the initial TDFL is based, and argue that the it can be rationally justified. After that I will describe the features of the HP application. We will see that features of the HP application are not consistent with the assumptions of the initial TDFL. This is probably the case in almost every application of technical diagnosis. So the initial TDFL cannot be justified as a solution to the HP task. I then describe a number of revisions that finally result in a version of TDFL that can handle the features of the HP application. Finally I make conclusions and say something about future work.

## F.3 The modelling language

I have so far only tried to write down what I consider essential in these CML components. This is to reduce the complexity of the analysis process. In fact, you will not immediately recognize this as CML components. The focus of the description is on the task expression part of the task body, i.e. decomposition and control. At the lowest level of decomposition, which corresponds

[1] I wish to thank Helena Tallqvist of Infologics and Anders Hemström of Swedish Telecom for providing information about the HP application

to primitive inferences, there is a verbal description of the domain knowlege that is needed. My choice of where to stop decomposing is rather arbitrary. Some of the tasks could easily be further decomposed. The tasks are actually decomposed more than what is necessary for controlling the sequence of external actions, so there is more control in the models than what is desirable in a knowledge level model. No inference structures have yet been drawn, because they can be derived from the task bodies.

The present version of CML does not prescribe a language for the task decomposition and control structure. The idea of describing a generic way to accomplish a task as a decomposition of the task into subtasks is very similar to the problem solving interpretation of Horn clauses [Kowalski, 1979]. Instead of using pseudo-code, I therefore use the Horn clause subset of first order predicate logic to represent this task decomposition to reduce unnecessary ambiguity. A task can be described as a predicate. The input and output knowledge roles are represented as arguments of the predicate. Usually the input is the first argument and the output is the last. The set of subtasks are represented as the body (the antecedent) of a Horn clause where the head (the consequent) represents the task. If there are alternative decompositions, they are represented as different clauses.

Instead of adding a specific control structure, I use the procedural interpretation of Prolog [Sterling and Shapiro, 1986]. It simply means that you solve the subtasks one by one in the sequence they are written in the decomposition. It may be the case that the body of a subtask does not provide an output solution, in which case the subtask fails. A task may have more than one solution, i.e. more than one possible output value, e.g. if there is more than one task body. If there is more than one task body, they are performed one at a time in the order in the which they are written, until the task is solved. If a subtask fails, you go back to the previous subtask that produced the input to the failed subtask and try to produce a new input to retry the failed subtask, etc. In order to specify iteration, we must then use recursive decompositions. I have found it convenient to extend the Horn clause language with an if-then-else construct. If the precondition subtask succeeds, you continue with the postcondition subtask, otherwise you continue with the subtask in the else-part. If the postcondition fails, you may go back and try more solutions of the precondition.

## F.4 Initial version of TDFL

The initial version of TDFL is based on my interpretation of Systematic Diagnosis by Localization [Breuker et al, 87], [Wielinga et al, 92]. I have made some immediate modifications to reduce the differences to the revised versions that I will present subsequently. What is said about this initial version is therefore not necessarily true for the original Systematic Diagnosis by Localization. If anything, my initial version is more restricted than then original version.

The top level task is called *troubleshooting*. The task body is defined in figure F.1, which also contains the task bodies of the subtasks. The subtasks which are not further decomposed are defined verbally below. The input to the troubleshooting task is a reference to a device. There is no information about what is wrong with it. The device has a number of distinct functions that can be used independently. The HP device can for example be used to control the amplitude and frequency of an output signal. It can also be used to measure amplitude and frequency of an input signal. The first thing to do is to find out which function is faulty. Each function is tested until one is found to be faulty, which is when the test subtask succeeds. We have then detected a specific fault. Diagnosis is the task of localizing the fault to the smallest possible level. We do that iteratively to a lower and lower level of detail. I distinguish between topDownDiagnose, which is the task to find the most detailed description of the fault, and diagnose, which is the task to find a more detailed description of the fault. The test task takes a part as input, which plays the role of hypothesis. The hypothesis is that the part is faulty. The first subtask is to find out the norm for the part. We then need to know what observations must be done to determine if the norm is satisfied. This is basically an instruction to the user. The next subtask is a transfer task. The user makes the required observations, which is a description of the actual behaviour of the part. If the actual behaviour is different from the specified norm, the part is faulty, and the test has succeeded. The topDownDiagnose task is simply the transitive closure of the diagnose task. The first step in the diagnose task is to decompose the part corresponding to the fault into a hypothesis part and the remaining parts which are called the rest. The hypothesis part is tested with the test subtask. If the test fails, we make the conclusion by elimination that the fault is in the rest.

$$
\begin{array}{l}
troubleshooting(Device, LowestLevelFault) \leftarrow \\
\quad functionalPart(Device, FunctionalPart) \wedge \\
\quad test(FunctionalPart) \wedge \\
\quad topDownDiagnose(FunctionalPart, LowestLevelFault) \\
\\
test(Part) \leftarrow \\
\quad specify(Part, Norm) \wedge \\
\quad observables(Norm, Observables) \wedge \\
\quad observe(Observables, Observation) \wedge \\
\quad difference(Observation, Norm, Difference) \wedge \\
\quad nonempty(Difference) \\
\\
topDownDiagnose(Fault, LowestLevelFault) \leftarrow \\
\quad diagnose(Fault, RefinedFault) \wedge \\
\quad if\ topDownDiagnose(RefinedFault, RefinedFault2) \\
\qquad then\ LowestLevelFault = RefinedFault2 \\
\qquad else\ LowestLevelFault = RefinedFault \\
\\
diagnose(FaultyPart, RefinedFault) \leftarrow \\
\quad decompose(FaultyPart, HypothesisPart, Rest) \wedge \\
\quad if\ test(HypothesisPart) \\
\qquad then\ RefinedFault = HypothesisPart \\
\qquad else\ RefinedFault = Rest
\end{array}
$$

Figure F.1: The initial version of task bodies used in TDFL.

$functionalPart(Device, Part)$: A device often consists of several distinct functions that are used individually. Each of those functions is a functional part and a solution to this subtask. This subtask is only applied to the top level device. The functional parts may be overlapping, in the sense that there may be a lower level part that is used to accomplish two functional parts of the device.

$decompose(SuperPart, SubPart, Rest)$: SuperPart is a part that consists of the two parts SubPart and Rest. There may be many ways of decomposing a part into two subparts, i.e. many solutions to this subtask. The two output parts may in turn be decomposed into parts in the same way. The domain model of the device does not have to be a binary decomposition. This task can be defined in terms of any kind of partOf-model. Rest may then be a reference to a set of parts.

$specify(Part, Norm)$: Norm is a set of necessary and sufficient conditions for Part to be functioning correctly

$observables(Norm, Observables)$: We have to look at the norm and see what observations need to be done in order to test whether the norm is satisfied. The Observables are the attributes which are referred to in the Norm.

*observe*(*Observables*, *Observation*): This is a transfer task, i.e. it is solved by an external action. The output observation is the actual observation made of the observables by the user.

*difference*(*A*, *B*, *Diff*): A and B are sets. If they are identical, Diff is empty. If they are different, Diff contains the elements in A that are not in B.

## F.5 Rational Justification

Although the above version of TDFL is extremely simple, it captures an important method used by service technicians performing the kind of task that the HP application is to support. They use other methods as well, but this one requires less experience of the particular device. In applications like HP, a service technician may never have seen the device before, but uses technical documentation to systematically decompose the device into parts and specify norms that can be tested. The term Systematic Diagnosis is in that sense quite suitable.
This version of TDFL has the attractive feature that we can rationally justify that it works.[2] Given a definition of the goal and a set of reliable assumptions, it is possible to prove that the task body works. One might argue that this should be required of a knowledge level model. The principle of rationality says that an agent takes an action if it knows that the action will lead to the goal [Newell, 82]. So this knowledge must be part of a knowledge level model. I use the term reliable assumption here in in the sense that we can expect a domain expert to know whether the assumption holds or not.

*Task definition* The input is a device. The output is a faulty part of the device. The goal is to find a faulty part of the device. It is desirable to locate a part at a fine level of detail.

*Assumptions* Domain knowledge assumptions: the agent knows every part of the device, and the parts of every part, and the agent knows the correct behaviour of each part. Transfer task assumption: if requested, the user will tell the agent the actual behaviour of any part. Case assumption: there is a fault in a part of the device. Method assumptions: a part behaves incorrectly if and only if there is a fault in that part, and faults are persistent, i.e. not intermittent.

*Justification* The agent will find a faulty part of the device, because one of the parts is faulty and the agent will find out which one by noticing the difference between actual behaviour and correct behaviour. By diagnosing the faulty part in turn, the agent will to the best of its ability locate the fault to a finer level of detail.

Note that we don't need to make a single-fault assumption. This is partly because the goal is only to find one fault. If there are more faults, you have to diagnose again. If we would have used a fault-model, the single fault assumption would have been necessary, because multiple faults may change the effects of each other. It is then the domain knowledge that is based on the single fault assumption, specifically that the agent knows the effect of a fault.
Other assumptions of Systematic Diagnosis have been mentioned in [Duursma, 1992]. For example that the cost of observing the behaviour of a part is equal for all parts. This assumption is necessary if a constraint is to keep down the cost of observations. You might argue that it is a tacit assumption of Systematic Diagnosis and the initial version of TDFL that the cost of observations should be kept down, because otherwise there would be no reason to decompose the device iteratively. If so, the task definition should contain this constraint. But if we do want to keep down the cost of observations, the agent must also have the ability to keep down the number of observations. I will get back to this after discussing the features of the HP task.

[2]The term rational justification was introduced in the context of validation by [Steels, 1991b]. There seems to be a relation to the notion of competence theories that has mentioned in some versions of D1.1.

## F.6 Features of the HP application

The following is a description of the features of the task and domain in the HP application. We will see that the initial TDFL is too simple to handle them. Observations must be made by the user, and they take time. One constraint is therefore to keep down the number of observations that need to be made in order to diagnose. More precisely, it is the total time required for observations that we want to keep down. Some observations take more time than others. A sometimes conflicting constraint is to choose a sequence of actions that the user will be able to understand. Another requirement is that the fault should be located to an appropriate level of detail. It should not be too low, because it requires more work from the user who makes observations, and it may be difficult to repair at a too detailed level.

We can distinguish some significant characteristics of the particular HP device, although they apply to most electronic devices. Most parts *cannot* be tested completely independently of the rest of the device. In particular you cannot control the inputs of a part with anything else than the external controls of the device. Although you can in principle force a signal directly on an internal conductor, you do not want to, because you may destroy something in the process. If you want to control an input of a component which is internal to the device, you have to input an external signal to the whole device and let it go through other components until it reaches the one you want. You have to make sure that the signal has not been corrupted on the way. Some parts are mounted so that they can be easily removed and tested independently, but that may also cause damage, and it is more the exception than the rule. Due to feedback loops, some inputs of some parts cannot be controlled at all, and hence those parts cannot be directly tested.

Recalling the assumptions of the initial TDFL, we must also make clear that the behaviour of an electronic device and its parts is too complex to be observed in its whole. It is however possible to specify norms that can be tested and are sufficiently strong to detect most faults.

### F.6.1 Requirements entailed by these task features

What do these task features entail for the initial set of TDFL components? We must change the definition of the task to include the constraint to keep down the total time required for observations. This means that we must add assumptions and change the task bodies to justify that this constraint is satisfied. In the following sections I present revisions of TDFL based on how the constraints are satisfied in the HP application.

In addition to the changed definition, one of the assumptions is now invalidated. This is the transfer task assumption that the user can tell us what the actual behaviour is. The problem is that you cannot test a part without first testing the preceding components in the signal flow. This means that some hypotheses are not directly testable. The final version of TDFL presented here handles this feature also.

We should also include in the definition the constraints to stop at an appropriate level of detail, and use a strategy that can be followed by the user. However, I have postponed this part of the analysis. Another feature which is not yet tackled is feedback loops that prevent testing.

## F.7 Reducing the time required for observations

We must first make assumptions to justify that we satisfy the constraint to keep down the total time required for observations. The only support that the initial version of TDFL provides is that the device is decomposed iteratively. This means that we never have to observe all parts at the lowest level of decomposition. By observing a composite part, we can exclude all the primitive parts within it. If we assume that it is cheaper to test a composite part than to test its parts, this will be a way to keep down the total time required for observations. However, it will be a very weak way.

The constraint to keep down the time for observations is not well defined. A judgement has to be made in each application about how much needs to be done to reduce the time required for observations. Again, iterative decomposition is in itself a very weak way – so weak that it will never be enough. In practice we make tacit assumptions about the decomposition model of the device, e.g. that a top level function is not decomposed directly into primitive parts. In practice, the decomposition model must be tailored to diagnosis if TDFL is to satisfy the needs to reduce the time required for observations.

There are more enhancements that need to be made in order to reduce the time required for observations to the extent needed in the HP application. We can restrict the norms to a representative sample of input stimuli, at the risk of failing to detect some faults. Furthermore, if we have detected a fault in the behaviour of a device, we do not need to test the whole behaviour of the parts when we try to localize the fault. It is enough to restrict the subsequent diagnosis procedure to a subset of input stimuli that result in faulty response.

The number of observations can also be reduced by recognizing the effects of some specific faults. If we can associate the actual fault with a specific cause, then the corresponding hypothesis is more likely than other hypotheses, and we should test it first.

There may be other reasons for preferring one hypothesis over the rest. The static, context independent, reasons can be incorporated in a static decomposition model.

The first requirement, that the decomposition model must be tailored to diagnosis, is significant but it can be fixed simply by specifying what that means. The same goes for the assumption that the norm can be restricted to a representative sample of the behaviour.

The requirement that the fault localization process must be restricted to the faulty segments of the behaviour, requires changes to the task body. Those changes will also allow us to recognize the effects of some specific faults, and use causal association to select more likely hypotheses.

## F.8 Decomposition model tailored to diagnosis

By decomposition model I mean the domain knowledge that is used to realize the tasks functional-Part and decompose that have been used in the initial version of TDFL. It is evident that we have to make some assumptions about the decomposition model used by TDFL, if we want to satisfy the constraint to keep down the total time required for observations.

If we assume that the level of primitive components is given, the decomposition of a device into primitive components is well defined. However, in iterative decomposition we want to introduce intermediate levels of decomposition – composite parts. The choice of what to regard as a composite part is arbitrary, unless we provide a more constrained definition. The concept of composite part in itself does not even imply a hierarchical decomposition, although there are reasons to prefer such a decomposition.

Although the choice of decomposition model is not well defined, there are some natural foundations. One of them is the decomposition made in the design of the device. Another is the physical configuration. A composite physical part must be such that it can be physically removed as a whole from the other parts. Another kind of constraint on a composite part may be that the subparts should be functionally part of the composite part. In diagnosis we want a functional decomposition, since it is the functions we are testing, and we want primitive parts that are physical parts, so that they can be replaced or repaired. However, those constraints are not enough. If it is cheaper to test a composite part than it is to test its subparts, then there is reason to include it in the decomposition.

Another important tacit assumption about the decomposition model is that the parts have inputs that can be controlled and outputs that can be observed during diagnosis.

Ideally we should provide a task body that creates a good decomposition for diagnosis on the basis of less task dependent device models. However, I think it is difficult, since there are so many factors to take into account. Until that, we have to assume that a service technician can develop a decomposition that is well enough tailored to diagnosis. I think it is a reasonable assumption, at least if we provide some guidelines.

## F.9 Detecting characteristics of the fault

The test subtask in the initial version of TDFL only results in success or failure. In the test task body, we actually find out more. We find out the actual behaviour of the faulty part. This knowledge can be used to reduce the time required for observations in the subsequent process of finding the lowest level fault.
In section F.7 I mentioned two ways of reducing the effort required for observations on the basis of the actual fault:

- we can restrict the subsequent process of diagnosis to a subset of the faulty segments of the behaviour.
- we can recognize the effects of some specific faults, and use causal association to select more likely hypotheses.

We can incorporate the above changes by modifying the task body of the task diagnose. The changes require that the input to diagnose is not just a reference to a faulty part, but a reference to the actual faulty behaviour of that part. Since the output of diagnose becomes the input to a subsequent invocation of diagnose, the output must also be a reference to the actual faulty behaviour of the lower level part.

### F.9.1 Fault as the actual faulty behaviour

We must first modify the test subtask, to output the actual observed faulty behaviour. I prefer to replace the test subtask with three subtasks. The first is the same specify subtask that was used in the task body of test. The other two are called *establish* and *consistent*. We must first make the notion of behaviour a little bit more precise by elaborating the verbal definition of the subtask specify.

$specify(Part, Norm)$: Norm is a set of necessary and sufficient conditions for Part to be functioning correctly. It is a description of the correct behaviour of a part of a device. It is described as a discrete mathematical function. It is described as a set of tuples where each tuple is divided into input values and output values. We therefore call them i/o tuples.

One more thing needs to be said about the notion of behaviour. The subtask decompose in the task body of diagnose should now take the faulty behaviour of a part as input. The output should be hypotheses about the behaviour of lower level parts. The notion of negative behaviour is introduced in the following revised definition of the subtask decompose. Since it is a slightly different task defined in terms of the previous version of decompose, I will give it a new name, decompose1. As we will see later, it would become awkward to assign meaningful names to refined versions of the task.

$decompose1(Fault, Hypothesis, Rest)$: Fault is the behaviour of a part. The part corresponding to Fault is decomposed using the previous decompose subtask into the parts corresponding to Hypothesis and Rest. Both Hypothesis and Rest are statements about behaviours. They state that the actual behaviour of the part they refer to, in the sample subset corresponding to the faulty subset of the input Fault, is incorrect. More precisely, the statement is the negation of the statement that the part has the a specific behaviour, which is the correct behaviour. We call this a negated behaviour. If the rest consists of a set of parts, the statement is the negation of the statement that they all have a specific behaviour, which is their respective correct behaviour.

The establish subtask is accomplished by the task body in figure F.2. The input is a hypothesis, a statement that needs to be verified. It refers to a specific behaviour. The output is the actual behaviour corresponding the hypothesis.

$$
\begin{aligned}
&establish(Hypothesis, Observation) \leftarrow \\
&\quad observables(Hypothesis, Observables) \wedge \\
&\quad observe(Observables, Observation)
\end{aligned}
$$

Figure F.2: The task body of the task establish.

$observables(Statement, Observables)$: Statement is a behaviour, or the negation of a behaviour. We have to look at the norm and see what observations need to be done in order to test whether the statement is satisfied. The Observables is the instruction to observe the actual i/o tuples corresponding to the input values in the behaviour referred to in the Statement.

$consistent(Hypothesis, Observation)$: This test succeeds if the two expressions are consistent. If both parameters are behaviours, the test is performed by an equality test. If the hypothesis is a negated behaviour, the test succeeds if the negated behaviour is different from the behaviour in Observation.

The revised version of the task body of troubleshooting is described in figure F.3. The subtask test is replaced by the subtasks specify, establish, and consistent. The fault input to topDownDiagnose is the actual behaviour of the faulty functional part.

$$
\begin{aligned}
&troubleshooting(Device, LowestLevelFault) \leftarrow \\
&\quad functionalPart(Device, FunctionalPart) \wedge \\
&\quad specify(FunctionalPart, Behaviour) \wedge \\
&\quad establish(Behaviour, ActualBehaviour) \wedge \\
&\quad not\ consistent(Behaviour, ActualBehaviour) \wedge \\
&\quad topDownDiagnose(ActualBehaviour, LowestLevelFault)
\end{aligned}
$$

Figure F.3: The task body of troubleshooting adapted to hypothesis as fault.

Before we can describe the revised task body of diagnose, we must introduce a new subtask similar to establish. The reason is that we have to derive the actual behaviour of the rest after refuting the hypothesis, in the step where we make a conclusion by elimination. The new subtask is called *establish* too, but takes previous observations as additional input argument, i.e. the actual (correct) behaviour of the hypothesis part. I only give it a verbal description, since it does not make use of any transfer task, and since no control is needed.

$establish(Hypothesis, Observations, ActualBehaviour)$: The test succeeds if the observations required to establish the hypothesis have been made in Observations. If so, the relevant observations are put in the ActualBehaviour parameter. The way it is used in diagnose, Observations will contain the actual behaviour of two parts, one part containing both the part corresponding to hypothesis and the other part. Every attribute of the hypothesis will therefore be observed in the Observations. If the other part on the same level in Observations is functioning correctly, it will be possible to derive the behaviour of the Hypothesis.

In the version of the diagnose task body in figure F.4, the fault is the actual behaviour of the faulty part. Since the fault is a behaviour rather than a part, it does not make sense to decompose it into parts. Instead the fault is decomposed into hypotheses about faults of parts. The hypothesis is then a statement that can be verified. If the hypothesis is refuted, i.e. if it is not consistent with the actual behaviour, we make the conclusion by elimination that the part corresponding to Rest is faulty. Since the part corresponding to fault consists only of the hypothesis part and the rest part, the actual behaviour of rest can be derived from the actual behaviour of the faulty part and the hypothesis part.

$$
\begin{array}{l}
diagnose(Fault, RefinedFault) \leftarrow \\
\quad decompose1(Fault, Hypothesis, Rest) \wedge \\
\quad establish(Hypothesis, ActualBehavior) \wedge \\
\quad if\ consistent(Hypothesis, ActualBehavior) \\
\quad\quad then\ RefinedFault = ActualBehaviour \\
\quad\quad else\ establish(Rest, [ActualBehaviour, Fault], RestBehaviour) \wedge \\
\quad\quad\quad RefinedFault = RestBehaviour
\end{array}
$$

Figure F.4: The task body of diagnose with the actual behaviour as fault

## F.10 Restricting diagnosis to a subset of faulty behaviour

The two changes mentioned in section F.9 can be made now that the fault is a behaviour. They both require different modifications to the decompose task body, i.e. the method of generating hypotheses. The most important improvement is to restrict the hypotheses, and thus subsequently the observations, to the faulty sample of the behaviour. The difference between the norm and the actual faulty behaviour tells us which sample of the behaviour that is faulty, and we can restrict the test of subsequent hypotheses to that faulty sample. In most cases we can restrict the subsequent observations to only one input stimulus. A new version of decompose is described in figure F.5, that uses two subtasks called *select* and *corresponding* to restrict the hypotheses to a subset of the faulty tuples in the behaviour.

$$
\begin{array}{l}
decompose2(Fault, HypothesisSubset, RestSubset) \leftarrow \\
\quad decompose1(Fault, Hypothesis, Rest) \wedge \\
\quad select(Fault, FaultSubset) \wedge \\
\quad corresponding(FaultSubset, Hypothesis, HypothesisSubset) \wedge \\
\quad corresponding(FaultSubset, Rest, RestSubset)
\end{array}
$$

Figure F.5: The task body of decompose that restricts the hypotheses to a subset of the behaviour.

$select(Fault, FaultSubset)$: A subset of the faulty tuples in Fault are selected and dedicated as FaultSubset. We first have to identify the faulty tuples, which is done by specifying the norm and comparing it with Fault using the difference test. ¿From the faulty tuples, a subset is selected. This is the subset that will be used in trying to locate the fault. Normally it will consist of only one tuple.

$corresponding(FaultSubset, Hypothesis, HypothesisSubset)$: The part corresponding to Hypothesis must be a part of the part corresponding to FaultSubset. HypothesisSubset consists of the sample of Hypothesis that corresponds to the sample FaultSubset. How do we define corresponding sample? The FaultSubset behaviour is accomplished by the behaviours of the parts of the part corresponding to FaultSubset. For each tuple in the FaultSubset behaviour, there is a corresponding tuple in the behaviours of the parts, that is used to carry out this particular tuple of FaultSubset. This can either be primitive domain knowledge, a relation between tuples, or it can be derived using knowledge of how attributes are related. If the hypothesis behaviour has input attributes in common with the FaultSubset behaviour, we just select the tuples where the values are identical. If not, we have to find out which part is in- between, etc. Actually, the relation between tuples is about the correspondence between input attributes. This means that it does not matter if the behaviours are fault or correct behaviours – the input values are the same.

## F.11 Testing more likely hypotheses first

The next improvement is that more likely hypotheses can be detected by making causal associations, and thus generated before less likely hypotheses. The new task body of decompose is defined in terms of the previous version, with the additional constraint to use the task *preferInSituation* to first select a likely hypothesis. The task bodies of decompose and preferInSituation are described in figure F.6.

$$
\begin{aligned}
&decompose3(Fault, Hypothesis, Rest) \leftarrow \\
&\quad preferInSituation(Fault, Hypothesis) \wedge \\
&\quad decompose2(Fault, Hypothesis, Rest)
\end{aligned}
$$

$$
\begin{aligned}
&preferInSituation(Fault, Hypothesis) \leftarrow \\
&\quad partOf(Fault, Hypothesis) \wedge \\
&\quad causes(Hypothesis, Effect) \wedge \\
&\quad consistent(Effect, Fault) \\
&\quad \vee \\
&\quad directPartOf(Fault, Hypothesis)
\end{aligned}
$$

Figure F.6: The task body of decompose based on selecting the most likely hypothesis, and the task bodies of its subtask preferInSituation

We have to distinguish between parts in general and the direct part in a tree-like decomposition. This is a distinction which was previously hidden in the verbal description of the first version of decompose.

$directPartOf(SuperPart, SubPart)$: This relation means that SubPart is a direct part of SuperPart in the decomposition model of the device. For simplicity, SuperPart and SubPart may also refer to a behaviour corresponding to that part

$partOf(SuperPart, SubPart)$: This is the transitive closure to the relation directPartOf. In other words, if X is a direct part of Y, then X is also a part of Y, and if X is a part of Y, and Y is a direct part of Z, then X is also a part of Z.

$causes(Fault, Effect)$: The relation means that the fault always results in the effect, if there are no other faults that influence the effect. This causal relationship is limited to those causal relationships that a service technician would use to override the more systematic diagnosis by decompose.

It is limited to a few well known causes with very specific effects. For example, in the HP application one specific component has the property that if it breaks, the amplitude of the output signal of the device will be reduced by half. Fault is the behaviour of a part. It is typically negatively defined, i.e. the statement that the norm of the part is not satisfied. The part corresponding to Fault must be a part of the part corresponding to Effect.

It is unfortunately not enough to refine the decompose step. The problem is the previous modification of decompose, which restricts hypotheses to a subset of the behaviour. The whole behaviour of the top level fault is only input to decompose in the first invocation of diagnose. In subsequent invocations of diagnose, it is the restricted sample of the faulty behaviour that is input. If more than one hypothesis can be causally associated with the top level fault, that information is lost after the first invocation of diagnose. The concluded fault must therefore contain the previously detected fault. This modification is made in figure F.7. The conclusion RefinedFault is a more and more detailed description of the actual faulty behaviour of the device. This is the model we build up of the device, i.e. the case model.

$$\begin{array}{l}
diagnose(Fault, RefinedFault) \leftarrow \\
\quad decompose3(Fault, Hypothesis, Rest) \wedge \\
\quad establish(Hypothesis, ActualBehavior) \wedge \\
\quad if\ consistent(Hypothesis, ActualBehavior) \\
\quad\quad then\ RefinedFault = [ActualBehaviour, Fault] \\
\quad\quad else\ establish(Rest, [ActualBehaviour, Fault], RestBehaviour) \wedge \\
\quad\quad\quad RefinedFault = [RestBehaviour, Fault]
\end{array}$$

Figure F.7: The task body of diagnose with a refinement sequence of behaviours as fault

## F.12 Hypotheses that are not directly testable

The next application feature to deal with is that a part cannot be tested unless the preceding parts in the signal flow are correct. What happens is that we cannot do establish of the selected hypothesis unless it is testable. This means that the assumption that the user can observe the behaviour of any part is violated. We have to introduce a subtask to determine testability. The testability of a part is dependent on what previous observations have been made. If the selected hypothesis is not testable, we have to use the method of test by elimination, which is based on finding a subsuming hypothesis that is testable, and begin by testing that, etc.
I will begin by explaining the problem in more detail. Imagine that we have located the fault to a block of components that are connected together in a single signal chain, as in figure F.8.

```
                        (components)
                       /   |   |   \
                      v    v   v    v
  input signal                           output signal
-------------->----[]---[]---[]---[]------------>---
```

Figure F.8: Four components in a signal chain

The only component that we can test directly is the first component in the chain, the one that has the same input as the chain. For any other component, the input will be unexpected if the

fault is earlier in the chain. Another way to put it is that we test a component by measuring its outputs. We must already know the input to a component when we test it. This means that we cannot generate hypotheses in the same random way as before. It would be relatively easy to modify the decompose step so that hypotheses are generated in the order dictated by the signal flow. However, this does not work if we want to give priority to more likely hypotheses, e.g. those causally associated with the actual fault. Unless the hypothesized part is first in the signal flow, the hypothesis will not be directly testable.
Let us assume again the example of a fault located to a chain of four components. The hypothesis is that the third component in the signal flow is faulty. What we do then is measure the output of the third component, knowing the input to the chain. If that observation is ok, we have refuted the hypothesis. If it is not ok, we have still not confirmed the hypothesis. We must also measure the input to the third component. If that is ok, we have confirmed the hypothesis, and we may try to refine it. However, if the observation is not ok, we have confirmed another hypothesis, namely that the fault is in the 1st or 2nd component.
A problem with describing a method for testing a hypothesis that is not directly testable is that the process of testing one hypothesis may result in other hypotheses being confirmed or rejected. A natural way to deal with this is to describe it as a method of test by elimination. We have to identify other hypotheses that can be tested and that can be used to make a conclusion by elimination about the desired hypothesis. You might say that we have to make a plan. In this application it is enough to identify two auxiliary hypotheses. The first one is the subsuming hypothesis. It is the smallest part that both contains the desired part and is testable. This will be a part that shares the outputs with the hypothesis part and which inputs are also inputs of the faulty part, and therefore have been observed. The next auxiliary hypothesis is the rest of the subsuming part minus the desired hypothesis part. It will be testable since its inputs are also inputs of the faulty part.

### F.12.1 Generating auxiliary hypotheses for elimination

Then we arrive at the final formulation of diagnose in figure F.10. It introduces two subtask called *testable*, which is verbally defined below, and *generateElimination*, which task body is described in figure F.9, that takes the actual fault and the desired hypothesis as input, and generates intermediate hypotheses that are testable and can be used to establish the desired hypothesis by elimination.

$$
\begin{aligned}
&generateElimination(Fault, Hypothesis, LargerHypothesis, Rest1, Rest2) \leftarrow \\
&\quad decompose3(Fault, LargerHypothesis, Rest1) \wedge \\
&\quad testable(LargerHypothesis, Fault) \wedge \\
&\quad decompose3(LargerHypothesis, Rest2, Hypothesis) \wedge \\
&\quad testable(Rest2, LargerHypothesis)
\end{aligned}
$$

Figure F.9: The task body of the task generateElimination

$testable(Hypothesis, Fault)$: This test succeeds iff the input attributes of Hypothesis are also input attributes of Fault, and the input values of the tuples in Hypothesis are contained in the input values of Fault. The meaning of testable with a negatively defined fault is that the hypothesis would be testable if the fault had been observed.

The neat distinction between generate and test is disturbed in this version of the diagnose task body. It may be possible to remedy this by replacing decompose with a task that generates a test plan. In this version, you might say that the task body contains two predefined plans. An alternative

$diagnose(Fault, RefinedFault) \leftarrow$
$\quad decompose3(Fault, Hypothesis, Rest) \wedge$
$\quad if\ testable(Hypothesis, Fault)$
$\quad\quad then$
$\quad\quad\quad establish(Hypothesis, ActualBehavior) \wedge$
$\quad\quad\quad if\ consistent(Hypothesis, ActualBehavior)$
$\quad\quad\quad\quad then\ RefinedFault = [ActualBehaviour, Fault]$
$\quad\quad\quad\quad else\ establish(Rest, [ActualBehaviour, Fault], RestBehaviour) \wedge$
$\quad\quad\quad\quad\quad RefinedFault = [RestBehaviour, Fault]$
$\quad\quad else$
$\quad\quad\quad generateElimination(Fault, Hypothesis, LargerHypothesis, Rest1, Rest2) \wedge$
$\quad\quad\quad establish(LargerHypothesis, LargerBehaviour) \wedge$
$\quad\quad\quad if\ consistent(LargerHypothesis, LargerBehaviour)$
$\quad\quad\quad\quad then$
$\quad\quad\quad\quad\quad establish(Rest2, Rest2Beh) \wedge$
$\quad\quad\quad\quad\quad if\ consistent(Rest2, Rest2Beh)$
$\quad\quad\quad\quad\quad then\ RefinedFault = [Rest2Beh, LargerBehaviour, Fault]$
$\quad\quad\quad\quad\quad else\ establish(Hypothesis, [Rest2Beh, LargerBehaviour], HypothesisBeh) \wedge$
$\quad\quad\quad\quad\quad\quad RefinedFault = [HypothesisBeh, LargerBehaviour, Fault]$
$\quad\quad\quad\quad else$
$\quad\quad\quad\quad\quad establish(Rest1, [LargerBehaviour, Fault], Rest1Behaviour) \wedge$
$\quad\quad\quad\quad\quad RefinedFault = [Rest1Behaviour, Fault]$

Figure F.10: The diagnose task body adapted to handle hypotheses that are not directly testable.

solution would be to only generate testable hypotheses. To get the desired behaviour with non-testable hypotheses, the decompose task would effectively in each invocation have to recognize that it should generate the next hypothesis in the plan. That would disturb the declarative character of the decompose task.

## F.13 Conclusions

The work presented here could serve as a detailed case-study for investigating principles for refining and modifying generic methods to the features of a real application. The following are some conclusions to that end. I have presented different versions of task bodies to reach a very similar goal. It is tempting to describe them as having the same task definition with different assumptions or features. In the Horn clause representations that I have made, these task feature distinctions have not been made explicit. In the case of the task decompose, I needed to represent the difference somehow, because one task body was defined as a specialization of the same task. I did that by giving the task a different name, by adding a numerical suffix to the name. In general that leads to redundancy, with different feature combinations. The modelling language should provide a way to make these specialization relations explicit. A task body that handles a specific feature, should be defined in terms of the task regardless of features. There is clearly a similarity to how specializations are made in object oriented programming.
Unfortunately, the distinction between task definition and assumptions and features does not seem to be well defined. As a way to capture commonalities and differences between task bodies, the distinction is relative to a set of task bodies.
If we want to provide guidelines for this kind of development process, it is necessary to describe these kinds of refinements and modifications as applications of more general problem solving methods.

The whole method of TDFL appears to be based on the same establish-refine regime as Hierarchical classification [Chandrasekaran, 1986] and top-down refinement [Steels, 90]. I think they can all be described as applications of binary search, in a general sense, which can be used when it is possible to determine with certainty in which part of the search space the solution is to be found. We must have a way to generate hypotheses that suits the estalish-refine regime. One way is to use a hierarchical decomposition model of the device. Another way is to use a method to create such a decomposition on the basis of other information of the device. The decompose task is also an application of abduction, based on the assumption $partOf(A, B) \wedge not(functioning(B)) \rightarrow not(functioning(A))$. Hypotheses are tested by empirical observation. If a hypothesis is negative, the observation tries to confirm the negated statement. The observation directly confirms or refutes the hypothesis. Although the observation in practice involves manipulation of external device controls, we can abstract away from that. Some hypotheses cannot be directly tested, in which case they are tested with a plan for test by elimination which is based on a sequence of the more straightforward empirical test.
The use of rational justification [Steels, 1991b], as exemplified with the intial version of TDFL, appears to be a powerful technique for validation, and for guiding the process of adapting a strong problem solving method to the features of an application.

## F.14 Future work

The description needs to be more adapted to CML. Furthermore, the analysis must be validated by respecifying the HP KBS as an application of the modelling components proposed here. This will also give empirical input to the work on capturing more aspects of the application in generic modelling components. Such aspects include features of the application that are not yet explicitly tackled:

- more detailed analysis of criteria for decomposition, e.g. the constraints to stop at an appropriate level of detail, and use a strategy that can be followed by the user
- feedback loops that prevent testing
- the method of testing physical components by physically removing them, e.g. by replacing them

It would also be interesting to continue the analysis from the point of view of basic modelling components.

# Appendix G

# Reverse Engineering of QUAESTOR with KADS

**Jan Top and Stephan Westen**

ECN

In this appendix we describe the conceptual analysis of a knowledge based system for configuring mathematical models in the field of naval engineering, called QUAESTOR. We describe the domain, inference and task structure of the conceptual model that was constructed to describe the expertise underlying QUAESTOR. We expect this analysis to play a role in the filling of the library with respect to the synthetic task of model construction. For that purpose this analysis will be restructured in terms of the CML-syntax whereas now a verbal description is given. Moreover, since the modelling task will be covered in another field as well, we expect to be able to present a more generic analysis in the near future.

## G.1 Introduction

The field of naval engineering is largely dependent on empirical knowledge, put down in literature in terms of a large number of mathematical design rules. In preliminary ship design these rules or partial models have to be combined into a single overall model in order to estimate the values of required design parameters. This is a process of collecting the proper design rules from the literature, solving the resulting system of equations and checking the validity of the solution found. At MARIN, the Dutch institute for naval architecture research, the knowledge based system QUAESTOR has been developed to support this process [van Hees, 1991a; van Hees, 1991b], which until now was mainly performed manually.

QUAESTOR has two major functions. The first function is to store and supply a large number (several hundreds) of mathematical relations that connect many parameters. Every relation may be subject to one or more constraints, defining its range of applicability. The second function is to perform a number of actions based on these relations: configure a model, ask for additional information and come up with a validated answer.

The global working of QUAESTOR is as follows. The user specifies his problem by giving the set of parameters that have to be determined. Using a number of heuristics, the system selects a set of relations. These relations introduce additional unknown parameters, for which values will be asked

from the user. The user may or may not provide this information. The process is continued until a solvable set of equations is obtained. The system is then solved and checked for its validity by calculating the constraints. If the system is valid the problem is solved and the initial unknowns are determined; otherwise another set of equations has to be selected, or it has to be decided that no solution can be found.
In this paper we describe a KADS model [Wielinga et al., 1992a] of QUAESTOR, or rather of the underlying expertise.[1] First, we will present the domain, inference and task layer. Then a short discussion will be given about our experiences in performing reverse engineering with KADS.

## G.2 Domain Layer

The domain layer of QUAESTOR is relatively simple. It consists of only three types of concepts: parameters, relations and constraints. Moreover, relations and constraints are closely related and treated much in the same way.
Each of these domain concepts has a number of attributes:

- **Parameter**

  **value** unknown or real

  **init_value** real

  **determined_by** (usr, sys, any)

  Initially the value of all parameters is unknown. During the inference process their values are derived, if possible. A parameter always has an initial value, which is given by the knowledge base. The attribute determined_by indicates how the value of the parameter can be determined: only by the user (usr), only by the system (sys) or by both (any). Hence, if for a parameter this attribute has value sys, its value will never be asked to the user. These attributes are preset in the knowledge base, but can be modified by the inference process, for example to indicate that the user has made known that he is not able to provide the value of a parameter.

- **Relation**

  **has_constraints** constraints

  **use** (function, equation)

  **rejectable** boolean

  **rejected** boolean

  A relation may have one or more constraints associated with it. A relation of type function can only be used to determine the parameter on the left hand side, whereas an equation can be used to derive the values of parameters on either side. A number of relations can be rejected by the user. Empirical relations are mostly rejectable, whereas relations based on physical laws are not.

- **Constraint**

  **value** boolean

  **rejectable** boolean

  **rejected** boolean

  Some constraints can be rejected. This means that the (informed) user is allowed to use a relation outside its proper range of applicability.

[1] Note that we do not pretend to describe the complete functionality of the system: several features of QUAESTOR will not be considered here. We restrict ourselves to the main problem solving process.

## G.3 Inference Layer

The overall inference structure is presented in figure 1. A rectangle represents a *dynamic knowledge role* (DKR) and an oval refers to a *primitive inference*.

### G.3.1 Input and Output

The input to the KBS consists of a set of parameters (DKR goal_pars) for which a valid model has to be assembled. After the inference process the system gives a valid system of equations (DKR sys_rels) that determines the values of these parameters, the values of all unknowns in the system and the set of associated constraints (DKR's sol_pars and sys_cons).

### G.3.2 Description of dynamic knowledge roles

The following dynamic knowledge roles are used by the inference structure. Every DKR takes a subset of one of the concept types: parameters, relations or constraints. In other words, the knowledge roles are different *views* on the knowledge base.

**G.3.2.0.1** lib_rels The complete set of relations contained by the knowledge base.

**G.3.2.0.2** lib_pars The complete set of parameters, all occurring in the complete set of relations.

**G.3.2.0.3** goal_pars The set of parameters constituting the problem, extended with additional unknown parameters that are needed to solve the problem.

**G.3.2.0.4** input_pars The set of parameters for which values are asked to the user.

**G.3.2.0.5** one_rels The set of relations that have only one unknown.

**G.3.2.0.6** calcd_pars The set of parameters for which a value has been derived directly (i.e., by direct substitution) from the values given by the user.

**G.3.2.0.7** sys_rels The set of relations that constitutes the system of equations.

**G.3.2.0.8** sys_cons The set of constraints associated with the set of equations.

**G.3.2.0.9** sol_pars The set of goal parameters, but with values that were calculated using the system of equations. Since the constraints of the system are not yet checked, some of these values need not be valid.

**G.3.2.0.10** prun_rels The subset sys_rels of which the relations that appeared to have false constraints have been removed, as well as the relations that were selected after them.

### G.3.3 Description of primitive inference actions

Here we describe the primitive inference actions identified in the system.

**G.3.3.0.11** select_system Given the set of unknown goal parameters (goal_pars), the set of known parameters (calcd_pars) and a part of the previously derived system (prun_rels) a selection is made from the full set of relations (lib_rels). This is the most important and complex PIA. It selects a minimal system of equations that is needed to compute the values of the parameters specified by the user. This means that the selection is repeated until a system with as many equations (relations *and* associated constraints) as unknowns is constructed. The selection process builds on the previous selected system for which unvalid relations (and their consequences) have been removed. There are several heuristics involved in this selection process.

**G.3.3.0.12** select_onerels ¿From all relations (lib_rels) select those with only one unknown (one_rels), given the set of already determined parameters (calcd_pars).

**G.3.3.0.13** select_goalpars Initially the user selects a number of parameters from lib_pars in order to specify the problem. During the inference process the system extends this set (goal_pars) with additional parameters that occur in the system of equations and cannot be derived directly from the input values.

**G.3.3.0.14** prune_system Remove all equations from the system (sys_rels) that have caused constraints (sys_cons) to be violated, producing a partial system in prun_sys. In QUAESTOR a system is constructed as a tree consisting of parameters interleaved with relations that are invoked by those parameters. Pruning boils down to cutting away a branch from this tree.

**G.3.3.0.15** compute_sys Given the set of equations (sys_rels) compute the solution. A numerical procedure is used if after direct substitution (parameters given by calcd_pars) a system of simultaneous equations remains. The system is linearised and solved using a multidimensional Newton-Raphson procedure.

**G.3.3.0.16** compute_cons Now that a solution is found all constraints that are associated with the system (sys_cons) are calculated and given value false or true in calcd_cons.

**G.3.3.0.17** select_onerels ¿From the set of equations (lib_rels) select those which have only one unknown (one_rels).

**G.3.3.0.18** compute_onerels Solve the equations with a single unknown (one_rels) and report resulting values to calcd_pars.

**G.3.3.0.19** assign_val The user may assign values to input_pars, resulting in calcd_pars.

**G.3.3.0.20** select_inputs Select those parameters from goal_pars that can be asked to the user and were not asked before.

## G.4 Task Layer

The task structure is relatively simple. Three tasks can be distinguished. The first task specifies the set of input and goal parameters, the second task assembles a valid system given the set of goal parameters and the third task tries to derive as many values as possible directly from the input parameters.

**G.4.0.0.21 Specify input and goal parameters** This task comprises the dynamic knowledge roles and primitive inference actions in the left column of the inference structure presented in figure 1. It is invoked every time when sys_rels in the goal-driven inference task selects new relations. First, the resulting set of (sub)goal parameters is selected. Second, a subset of these parameters is given to the user in order to provide any additional information. Finally, the set of goal parameters is reselected in order to take into account the new data.
select_goalpars(lib_pars, sys_rels, calcd_pars) → goal_pars
select_inputs(goal_pars, calcd_pars) → inputpars
*perform task* data-driven inference
select_goalpars(lib_pars, sys_rels, calcd_pars) → goal_pars

**G.4.0.0.22 Goal-driven inference** This is the most important task. The user initialises and starts it by specifying a number of goal parameters This task includes the second and third column of the inference structure in figure 1. Task specify input and goal parameters is called twice: first to initialise the global inference process and second to specify (sub)goal parameters for each configuration cycle.

*perform task* specify input and goal parameters
*repeat*
    *repeat*
        select_system(lib_rels, prun_rels, calcd_pars) → sys_rels
        *perform task* specify input and goal parameters
    *until number of equations equals number of unknowns*
    compute_sys(sys_rels, calcd_pars) → sol_pars
    compute_cons(sol_pars, sys_cons) → sys_cons
    prune_system(sys_rels, sys_cons) → prun_rels
*until system valid or no more systems can be constructed*

**G.4.0.0.23 Data-driven inference** The third task performs forward reasoning: it derives new information from input data. It is invoked every time new values are supplied by the user and calculates as many new values as possible by direct substitution. Figure 1 shows the associated inference structure in the two columns on the right.

assign_val(input_pars) → calcd_pars
*repeat*
    select_onerels(lib_rels, calcd_rels) → one_rels
    compute_onerels(one_rels) →) calcd_pars
*until no more relations with one unknown can be found*

## G.5 Reverse Engineering

QUAESTOR is an existing KBS. It was designed and implemented by an expert in naval engineering, and the underlying expertise was translated directly into the actual implementation. In his own words, the expert has tried to imitate his own behaviour in dealing with this type of problem. No (explicit) conceptual or design model has been made, although it should be mentioned that the structure of the implementation is highly modular and a design model could be constructed from the implementation.

Due to the fact that the KBS was already implemented, the conceptual analysis of the underlying expertise was a matter of *reverse engineering*. When discussing the functionality of the system, the expert who built the system tacitly switched between naval engineering terminology like 'propulsive power, propeller pitch, thrust coefficient, etc.' and implementation details like 'network address, pointer, stack, etc.' However, the conceptual modelling had to abstract from these terms in order to describe the inference process at the proper conceptual level.

An important lesson learnt from this analysis is that the model has to be of a rather coarse grain size, instead of trying to represent all functionality present in the system. The most difficult job was not to abstract from typical domain knowledge about naval engineering, but to elicitate the actual inference structure that was implicitly employed by the expert and hidden in the programme. During the modelling process we gradually succeeded in breaking away from the implementation. If the model had been made exactly according to the implementation and the related comments of the expert, almost every statement should have been translated into some primitive inference action. Due to the restriction of having a limited number of meaningful abstract modelling items, some fundamental shortcomings of the system were clarified:

- The fact that the system actually has to perform a search process in a very large search space ($2^n$ possible models for $n$ submodels or relations) was not recognised by the expert. As a consequence, in QUAESTOR no record is kept of the models that have already been visited, and the system is trapped if it doesn't reach a solution within two inference cycles. However, the interesting point is that these problems only appear if the system is tested by constructing a simple knowledge base that has no meaning to any real domain. Apparently, the heuristics in QUAESTOR are very effective, but only in combination with the proper, informed knowledge base. There is so much knowledge contained implicitly by the knowledge base that in most cases it performs adequately.

- However, adequate performance is not only dependent on a well designed knowledge base, but also on proper control by the user, although less significantly. For example, sometimes submodels may be proposed to the user that deal with a completely different problem than the one considered, but happen to be mathematically related. The informed users will reject these submodels as being nonsensical, but others will not. In that case QUAESTOR may get stuck. This imposes a serious problem: how much knowledge is the user exactly expected to have? This is particularly important if the user is also allowed to modify the library of relations. The expert admits that this is a problem. He is interested in attempts to extend the system with the knowledge necessary to make domain dependent control decisions in order to remove that burden from the user.

Finally, some remarks have to made with regard to KADS as a guide line for constructing the conceptual model. First, although the level of detail is an important aspect of the modelling process — as we observed above — there is not much guidance in KADS with respect to this. Perhaps this should simply be considered as a part of the skills and experience of the knowledge engineer. Second, we have to mention the fact that an important aspect of the inference structure is not made explicit in the KADS model: the role of the concept attributes in controlling the action within each primitive inference action. Finally, to our surprise we ended up with only a few different types of PIA's, *viz.*, select, compute and assign_value, in agreement with those suggested by the KADS developers. To our knowledge there is no existing interpretation model that contains the inference structure for the synthesis task considered here. However, the *constrained modelling* process is not restricted to this domain or to mathematical modelling alone. The inference process is expected be to useful to other application areas as well.

## G.6 Conclusion

We have given a conceptual model of QUAESTOR, a knowledge based system for constructing mathematical models in the field of naval architecture. In line with KADS the domain, inference and task layer were described. Three tasks were distinguished: (i) specify input and goal parameters, (ii) assemble a valid model from the given set of required parameters and (iii) derive as much information as possible information from the input data.
The process of constructing a conceptual model from an existing application revealed some shortcomings of QUAESTOR, due to the fact that it abstracts from the implementation. As a consequence, the performance of the system strongly depends on the quality of the knowledge base and — in a less extent — of the familiarity of the user with the domain.
We also discussed the process of reverse engineering. We observed that it is not straightforward to find the proper level of granularity for the conceptual model. Further, the inference structure presented here can be typical for other constrained modelling tasks.

# Appendix H

# Analysis of Assessment tasks

**Christiane Löckenhoff**

SIEMENS

This appendix is a first attempt to give a description of the task of assessment in a bottom–up approach. This bottom–up approach is based on several papers about the task assessment which were submitted to the 2nd KADS User Meeting in Munich [Bauer, 1992]. The different inference structures for assessment were discussed in a T1.3.6 meeting the day after the workshop. The outcome of this discussion about what the general notion of assessment is will be outlined in this appendix.
Since the library structure as it is outlined in the main part of this deliverable has only recently been defined, the description of the task of assessment does not fit to this definition in all parts but is based on draft versions of D1.1. It has to be adapted to the library structure, but it has as well to be refined since what is written here is only a first attempt. Both will happen in summer '92.

## H.1 The Task of Assessment

Assessment is a task which has been analysed and modelled with KADS several times: five papers were submitted to the 2nd KADS User Meeting in Munich [Kingston, 1992; Knemeyer, 1992; Lockenhoff and Reminger, 1992; Lochak, 1992; Porter, 1992]), at least two assessment applications have been developed at Touche Ross [Porter, 1992].
Assessment in itself is a task in which a case is mapped onto a decision according to a measurement system. The case usually is described in terms of observables, either simply a set of them or a structure, following e.g. an application form. The decision is an element of an either continuous or discrete decision space. In the latter case the decision itself is often called a "decision class", whereas in the former it is often called "ranking". The measurement system is either a set of typical cases or an abstract criterion system.
Thus, the task definition for the task of assessment according to the CML defined in D1.1 [Wielinga et al., 1992a] is the following:

**task** *assess (C, D)*

**task definition**

**goal**

```
FIND a decision D for case C according
to the given measurement system M.
```

**case roles**

```
case user input data → C: case description
case solution → D: decision.
```

**spec**

D results from using M for assessing C $\bigwedge$
$\forall D^A 1$: decision
$D^A 1 \neq D \vdash D^A 1$ does not result from using M
for assessing C

The assumptions underlying this task definition are:

- *The measurement system M is proper for assessing C, i.e. there is a decision which results from using M for assessing C.*
- *On the domain layer, there is a structure which maps onto the* `case description`.
- *On the domain layer, there is a structure which maps onto the* `decision`.

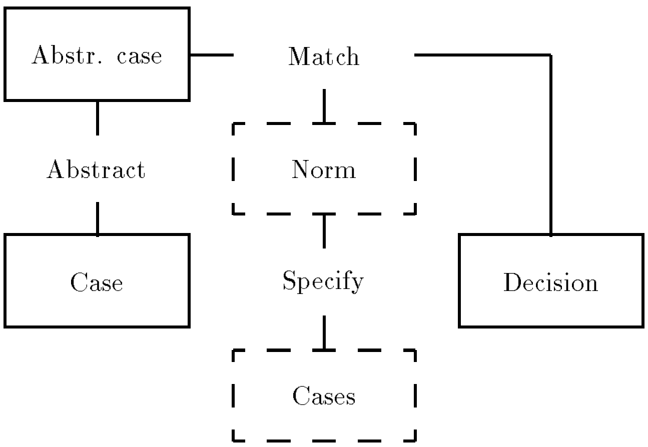
As one can easily see, the last two assumptions refer to a generic structure of the domain layer, whereas the first assumption refers to the modelling in general.
In analysing the various model for assessment we found that there are two different approaches to assessment, a case–based approach and a theoretical aproach. According to these two approaches there are two different task bodies which will be defined in the next two subsections.

### H.1.1 The Case–based Approach

In the case–based approach, the measurement system consists of a set of cases. These cases are typical according to the decision space, i.e. they are either "good" (ideal) or "bad" (fault) cases. The number of the cases may range from one to many.
The task of assessment then can be broken down into the following subtasks:

- Abstract the case description into an abstract case description.
- Specify a norm out of the set of typical cases.
- Match the abstract case description against the norm.

Put in a graphical form, the case–based approach has some similarity with the heuristical classification schema by Clancey [?]:



Note that the decision plays the role of the solution in Clancey's view whereas the dashed column in the center does not exist in Clancey's heuristic classification.
The task body for assessment in the case–based approach is the following:

**taskbody**

**assumptions**

. . . not known yet . . .

**subgoals**

```
G1: ABSTRACT case C
G2: SPECIFY norm N
G3: MATCH norm N and abstract case A
```

**subtasks**

```
G1: abstract (C, A)
G2: specify (M, N)
G3: match (N, A, D)
```

**control–roles**

**abstract case** `A: abstract case description`
**norm** `N: abstract case description`

**task expression**

```
abstract (C, A)
specify (M, N)
match (N, A, D)
RESULT (D)
```

For using the case–based approach in an assessment application, the following assumption has to hold for the application:

*There has to be at least one typical case which has already been assessed.*

Since it may happen, that the abstract case influences the specification of the norm (because e.g. the set of information per case may differ) or that the specified norm influences the abstraction process (because e.g. only a subset of the case information is needed for being matched with the norm), it is useful to allow for the following modifications, which can be used only exclusively for not getting circles:

- The `specify` subtask may have the abstract case A as additional input: `specify (M, A, N)`.
- The `abstract` subtask may have the norm N as additional input: `abstract (C, N, A)`, in which case the first two lines of the task body expression have to be swapped.

### H.1.2 The Theoretical Approach

In the theoretical approach, the measurement system consists of a system of abstract criteria. The system could e.g. be a hierarchy. The entry point to this system is an abstraction of the type of cases under focus. This entry point can be seen as the main criterion whose value directly gives the decision for the actual case. The criteria in general are either to be assessed via sets of specific subcriteria or to be computed out of a set of observables.
Thus, the subtasks for assessment in this approach are:

- Decompose the criterion under focus, if decomposable.
- Select parameters according to the criterion to be computed.

– Get case data.
– Compute the value of a criterion out of the case data.

At this point there are two possibilities. Either one regards the criterion under focus as something given and not to be mentioned, then the task assessment can be described with a recursive task expression where determining the value of a subcriterion is seen as an assessment as well. In this case the case role `case solution` has not only to map onto the decision space but as well onto the value ranges of all the subcriteria, which in my opinion is not good modelling.
The other possibility is to define a task `assess_criterion` or the like, which not only has the actual case as input but also the criterion under focus, and which delivers the value of this criterion as output. This task then is not basic anymore, it is a generic task. The task expression for this `assess_criterion` has to be recursive as well but the parameter the recursion is about is now visible.
This point needs further thinking about before a CML description can be given.

## H.2 The Task Decompose

When regarded as a task, it is assumed that the process of decomposition is complicated to a degree that it has to be elaborated on and further decomposed, e.g. in case the order in which the components of a system are delivered is important. In this case, the subtasks of the task `decompose` are

– Determine the set of all components of the system to be decomposed (only one level of detail lower).
– Select one element of this set according to some priority rules.

The task definition then runs as follows:

**task** *decompose (S, C))*

**task definition**

**goal**

```
FIND a component C of system S according
to the given priority rule system R.
```

**case roles**

```
case user input data → S: component
case solution  → C: component.
```

**spec**

C is a component of S $\bigwedge$
$\forall$ C$^A$1: component (C$^A$1 is component of S $\bigwedge$ C$^A$1 $\neq$
C) $\vdash$ \verbC is not component of C/$^A$1

**taskbody**

**assumptions**

– –

**subgoals**

```
G1: DETERMINE set M of components of system S.
G2: SELECT one of the components.
```

**subtasks**

```
G1: find (S, M)
G2: select (M, C)
```

**control–roles**

**component–set** `M: set of components`

**task expression**

```
find (S,M)
select (M, C)
RESULT (C)
```

The assumptions for this task description are:

- *The system S is decomposable.*
- *The rule system R is proper, i.e. there can always be selected one component out of the set of components which fulfills the rules.*

## H.3 The Primitive Inference Decompose

When regarded as a primitive inference, the process of decomposition is seen as an easy one from the modelling point of view. The description of the inference `decompose` in CML then runs as follows:

**inference–relation** *decompose (S, C)*

**goal**

`GET one component C of the system S.`

**dynamic roles**

`S: system` → `system component`
`C: component` → `system component.`

**static roles**

`decomposition model: part-of (S, C).`

**inference–specification**

`part-of (C, S).`

Clearly, this inference is non-directional, since you can use it for decomposing as well as for composing systems.
The assumptions for using this pia are:

- *There is some concrete decomposition knowledge modelled on the domain layer.*
- *It is not important in which order the components of the system to be decomposed are delivered.*

The second assumption is the one which points out that the actual `decompose` should be seen as a primitive inference action. The first assumption which is reflected in the static role `decomposition model` has to hold for decomposition in general.

# Appendix I

# Analysis of Assessment tasks

**Christiane Löckenhoff**

SIEMENS

This appendix presents an inference structure for technology assessment in terms of KADS–I. It is taken from a paper by the same authors, previously published in the Proc. of the 2nd KADS User Meeting in Munich (see [Bauer, 1992]). The topic of this workshop was to get an overview over inference structures built for existing systems and to compare them. The inference structure, which describes the behavior of an expert, is a refinement of the KADS interpretation model for assessment [Breuker et al., 1987a]. It has to be remodelled in terms of CommonKADS which will happen in summer '92.

## I.1 Background of the Inference Structure

The most important criterion for modern technology planning is to come to a full consensus within the company concerning the profitable objectives of research and development as quickly as possible. At present such processes generally run very slowly and are uncoordinated, as they are not supported by unbiased methods and instruments. Therefore, a knowledge based system was built by which the future opportunities and the risks of a technology can be analyzed.

Primarily, such a knowledge based system (KBS) supports the formulation of goals by research and development (R&D) experts. It guarantees objectivity by a deeply structured analysis procedure. Using the KBS diminishes the time for technology evaluation. But the main advantage of the KBS is that strategic discussions within the management are placed on an objective level and are no longer characterized by eloquence and political power of individuals. In this respect the system does not replace entrepreneurial decisions but diminishes the risk of resource wastage.

Technology assessment consists of interpreting the data known about a specific technology and finding a proper decision class. Data are for example the number of patents, publications, and the amount of R&D funding. Decision classes are e.g. *changing project goals*, *increase financial ressources*, *ask business groups for more commitment*, and such like. In the interpretation process, some well-known methods of technology management like the S-curve or the technology life cycle concept (see [Reminger, 1990]) are used. By a check of plausibility it is ensured that there is no inconsistency in the results.

In the KBS the main criteria in assessing a technology are assessed by using specific subcriteria which have to be chosen depending on the technology and the availability of information. The main criteria and their values are then presented to the system's user for the classification of the technology. The KBS itself does not provide a classification in decision classes and was never meant to since technology assessment is a highly sensitive topic.
Regarded in this way, the KBS – which has been applied in practice now for more than one and a half year – only performs data interpretation. But according to the ideas of KADS [Wielinga et al., 1991] we regard the task to be fulfilled as assessment with the subtask data interpretation done by a knowledge based system as the agent and the subtask relating to a proper decision class done by a human being as the agent.

### I.1.1 Inference Layer

In this subsection we will explain our inference structure. According to the papers written about KADS, in an inference structure only knowledge sources may appear, but no transfer tasks. The latter are tasks that require interaction with an external agent, usually the user [Wielinga et al., 1991]. But leaving out the transfer tasks, how can external information be introduced into and used in an inference structure ? For in specifying an inference structure not for a library but for modelling a real life task one has to model much more exact where knowledge comes from and how.
What we did in developing the inference structure for assessment is – transferred to the task of diagnosis – the following: we introduce another knowledge source into the inference structure, named e.g. *get value*, and declare in the description of the knowledge source that it is to be realized as a transfer task of a specific type – in this case *obtain*. In the task decomposition tree this is then shown by putting a leaf *obtain* under the knowledge source *get value*.
Solving the problem in this way one can specify an inference structure in a grain size necessary for it being a specification of a knowledge based system. On the other hand the generality of an inference structure is not lost because in the structure nothing about how knowledge source are to be understood – regarding being a transfer task or not – is said.
Having this in mind, the inference structure we are now going to present can be understood properly.

### I.1.2 Inference Structure

For the sake of better readability the inference structure of the task assessment – as we see it – is distributed over several layers. The top layer is shown in fig. 1 where knowledge sources are written in ovals, metaclasses in rectangles, metaclasses from an upper layer of the inference structure hatched, and expandable knowledge source are surrounded by a bold line.
In this figure assessment is described as consisting of four major inferences. The first is to specify a term out of the system model. The second inference is finding a value for the selected term depending on the specific case data which is an assessment task in itself. With the third inference step all the terms are gathered together into an abstract description of the specific technology. The fourth inference step lies in relating this abstract description to an adequate decision class. Since technology assessment is a highly sensitive topic this inference – as outlined above – is to be carried out not by the KBS but by human experts.
Not every term has to be updated each time the user runs the system. In some aspects a technology which has been analyzed before may not have changed, others have been updated recently. So the user has to decide which terms have to be assessed (see fig. 2).
The structure of the inference step assess term is shown in fig. 3. Assessing a term is done in four steps: finding a set of subterms – which is a set of subcriteria – for the term, determining the value of a single subterm, comparing the values of the different subterms as a check for plausibility, and, in case plausibility is given, computing the value of the term out of the values of the subterms.

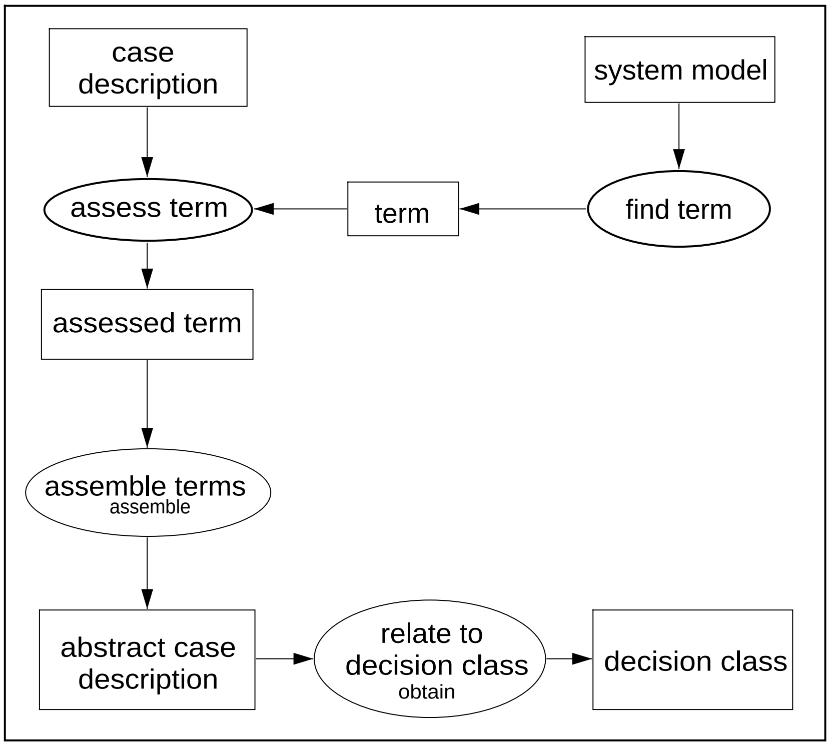



Figure I.1: Inference Structure for *assessment*

But if there are inconsistencies, the user has to be asked whether the subterms shall be assessed again or the assessment process as a whole shall be stopped.
Finding the proper subterms so that the main term itself can be assessed consists of finding out which subterms are relevant for the specific technology, and about which of these subterms information is available for the system's user (see fig. 4).
Assessing the value of a subterm (see fig. 5) can be done by three inference steps: specifying the parameters which have to be observed, obtaining their values in a dialogue with the system's user, and computing the value of the subterm by evaluating a mathematical expression. Which expression is to be applied depends naturally on the subterm to be computed, as well as the parameters whose values are known by the user (and the system).
Looking at the inferences structures above – and especially at the three figures 1, 3, and 5 dealing with assessing something – we can state that assessment in general is done in at least three steps: specifying one or more terms, finding values for them and relating them as a whole to an adequate decision class.

### I.1.3 Input and Output

The input problem is represented by the metaclass case *description*. This metaclass represents the information which is necessary and can be obtained for a technology in general, id est the set of all parameters which could be useful in assessing the technology. This information is not given from the very beginning but is asked for – if needed – during the runtime of the KBS.
The output of the KBS is represented by the metaclass *abstract case description*. It contains the fixed set of the ten main criteria together with their values for the analysed technology. The result of the whole task of assessment is the metaclass *decision class* which depicts the result of the assessment process.

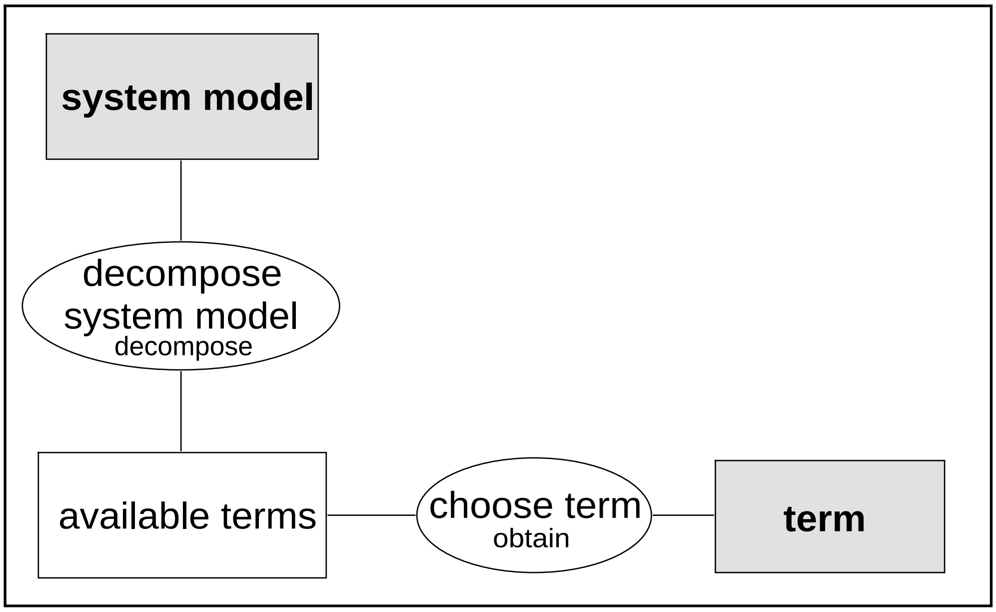



Figure I.2: Inference Structure for *find term*

In the following we will outline an example from the application:

> Every core technology within Siemens AG is evaluated two times a year on the basis of ten criteria. The first step is to select one of the ten terms to be updated, for example the term *potential for technological development*. The estimation of this potential has to rely on subcriteria like *Arthur D. Little's technology life cycle concept*, *patent applications*, *publications*, or the *model of industrial innovation* (see [Reminger, 1991]). The user is requested to select at least two of these subcriteria which are relevant for our technology and for which he has enough information.
>
> The next step, the assessment of the subterms, is done by specifying the observables which are qualitative indicators for the subterms and by obtaining their values. For example, indicators for the subterm *patent applications* are *activity*, *actuality* and *concentration of the patent applications*. These observables are then used in computing a value for the subterm *patent applications*.
>
> By doing a plausibility check between the subcriteria *technology life cycle concept*, *patent applications*, and *publications*, the KBS assures that there are no inconsistencies in the results. In case there are inconsistencies, the research manager and a member of the planning department have to find out the reason, e.g. a misunderstanding of some questions, or incorrectness of the input. If the findings are consistent, the subterms can be aggregated depending on their reliability: the value of the term *potential for technological development* is assessed.
>
> After all criteria have been assessed, the values are represented to the system's user in a diagram like the one in fig. 6. Having this diagram at hand, the managers may now relate the technology under consideration to a decision class.

### I.1.4 Description of Metaclasses

In table 1 we described the metaclasses in the order in which they appear in the inference structures in the respective section.

| Metaclass | Domain Specific Description | Data Structure |
|---|---|---|
| system model | The system model contains a description of all the criteria – be they subcriteria or main criteria – and the relationships between them. | structure |
| term | A term is a main criterion for the *assessment* of a technology, e.g. *missing alternatives*. The set of main criteria is fixed, it contains ten elements. | concept |
| case description | The case desctiption consists of all the parameters which can be asked for in the context of assessing a technology. | set of concepts |
| assessed term | Same as term, but with value | concept |
| abstract case descr. | The abstract case description consists of all the main criteria with all their values. | set of concepts |
| decision class | This metaclass represents all the possible decisions. | concept |
| available terms | The set of the ten main criteria. | set |
| subterms | This metaclasss represents all the subcriteria which are chosen for assessing a technology. | set of concepts |
| subterm | A subterm is a subcriterion which can be used for assessing a technology. | concept |
| subterm with value | Subcriterion with a value. | concept |
| subterms with value | resp. | set of concepts |
| plausibility | This metaclass represents the decision term for whether the values of the chosen subcriteria are plausible or not. | concept |
| ongoing | This metaclass represents the decision of the user how to go on in case of inconsistencies. | concept |
| available subterms | Available subterms are all those subcriteria which may be used in assessing a technology in general. | set of concepts |
| observables | Observables are the parameters / indicators which are of interest for assessing a specific subterm for a specific technology. | set of concepts |
| findings | Same as observables, but with values. | set of concepts |

Table I.1: Table of Metaclass-Descriptions

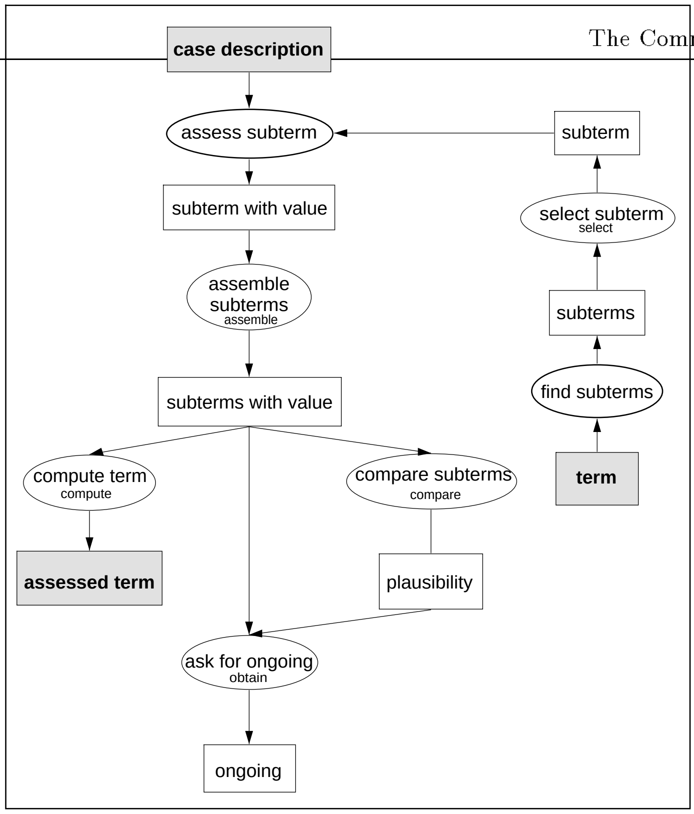



Figure I.3: Inference Structure for *assess term*

### I.1.5 Description of Knowledge Sources

In table 2 we describe in the order of their appearance the knowledge sources, their in– and output metaclasses, the methods used and the domain knowledge required in a somehow systematic way.

## I.2 Task Layer

How the task assessment can be decomposed in subtasks is shown in fig. 7. The leaves of the decomposition tree are either knowledge sources or transfer tasks. Transfer tasks are written in italics, decomposable tasks in roman, and knowledge sources in the normal style.
The task control structure is the following:

– **assess (decision class)**

decompose system model (system model → available terms)

*choose term* (available terms → term)

WHILE term is chosen

DO assess term (term → assessed term)

*choose term* (available term → term)

assemble terms (assessed term → abstract case description)

*obtain decision class* (abstract case description → decision class)

– **find term (system model → terms)**

decompose system model (system model → available terms)

*obtain term* (available terms → term)

| | |
|---|---|
| knowledge source | assemble terms |
| input | assessed term |
| output | abstract case description |
| method | All assessed terms are assembled in the abstract case description. |
| domain knowledge required | – |
| knowledge source | relate to decision class |
| input | abstract case description |
| output | decision class |
| method | Transfer task of type *obtain*: User has to relate the abstract case description to decision class. |
| domain knowledge required | – |
| knowledge source | decompose system model |
| input | system model |
| output | available terms |
| method | Decomposing relationship between technology and main criteria. |
| domain knowledge required | Relationship between technology and main criteria. |
| knowledge source | choose term |
| input | available terms |
| output | term |
| method | Transfer task of type *obtain*: User selects one term out of the set of available terms. |
| domain knowledge required | – |
| knowledge source | select subterm |
| input | subterms |
| output | subterm |
| method | Out of a set of subterms one subterm is to select which has not been assessed yet. |
| domain knowledge required | – |
| knowledge source | assemble subterms |
| input | subterm with value |
| output | subterms with values |
| method | All assessed subterms are assembled in a set. |
| domain knowledge required | – |
| knowledge source | compare subterms |
| input | subterms with value |
| output | plausibility |
| method | The plausibility of the subterms' values are checked. |
| domain knowledge required | Which subterms have to be compared, and tolerance limits for the different values. |

| | |
|---|---|
| knowledge source | ask for ongoing |
| input | subterms with value, plausibility |
| output | ongoing |
| method | Transfer task of type *obtain*: User says how to go on. |
| domain knowledge required | – |
| knowledge source | compute term |
| input | subterms with value |
| output | assessed term |
| method | The subterms are aggregated with a term–specific formula into one value. |
| domain knowledge required | Relationship between terms and formulas, and between subterms and their reliability. |
| knowledge source | decompose term |
| input | term |
| output | available subterms |
| method | Decomposing the relationship between terms and their subterms. |
| domain knowledge required | Relationship between terms and their subterms. |
| knowledge source | choose subterms |
| input | available subterms |
| output | subterms |
| method | Transfer task of type *obtain*: User delivers those subterms which are to be used further on. |
| domain knowledge required | – |
| knowledge source | specify observables |
| input | subterm, case description |
| output | observables |
| method | Using the relationship between parameters in the case description and the subterms. |
| domain knowledge required | Relationship between parameters in the case description and the subterms. |
| knowledge source | get values |
| input | observables |
| output | findings |
| method | Transfer task of type *obtain*: User delivers the values of the observables. |
| domain knowledge required | – |
| knowledge source | compute subterm |
| input | subterm, findings |
| output | subterm with value |
| method | According to the subterm a formula is applied to the findings to gain the value of the subterm. |
| domain knowledge required | Relationship between subterms and formulas. |

Table I.2: Description of Knowledge Sources

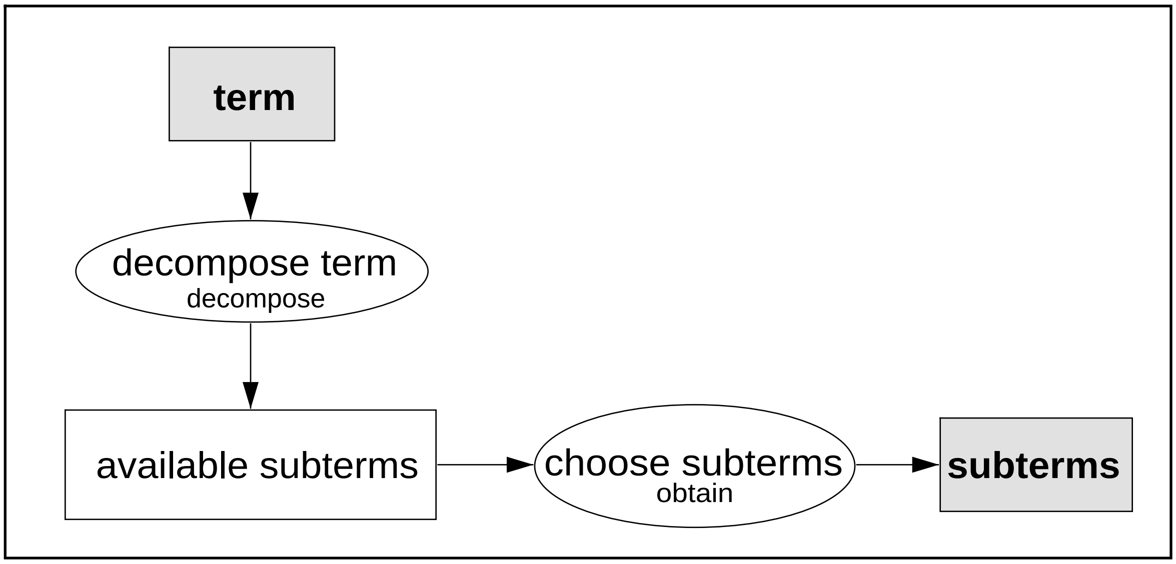



Figure I.4: Inference Structure for *find subterms*

- **assess term (term → assessed term)**

  find subterms (term → subterms)

  REPEAT select subterm (subterms → subterm)

  assess subterm ( subterm, case description → subterm with value)

  UNTIL each subterm in subterms is assessed

  assemble subterms (subterm with value → subterms with value)

  compare subterms (subterms with value → plausibility)

  IF plausibility is given

  THEN compute term (subterms with value → assessed term)

  ELSE *obtain ongoing* (subterms with value, plausibility → ongoing)

  IF ongoing is to stop

  THEN stop

  ELSE assess term (term → assessed term)

- **find subterms (term → subterms)**

  decompose term (term → available subterms)

  *obtain subterms* (available subterms → subterms)

- **assess subterm (subterm, case description → subterm with value)**

  specify observables (subterm, case description → observables)

  *obtain values* (observables → findings)

  compute subterm (subterm, findings → subterm with value)

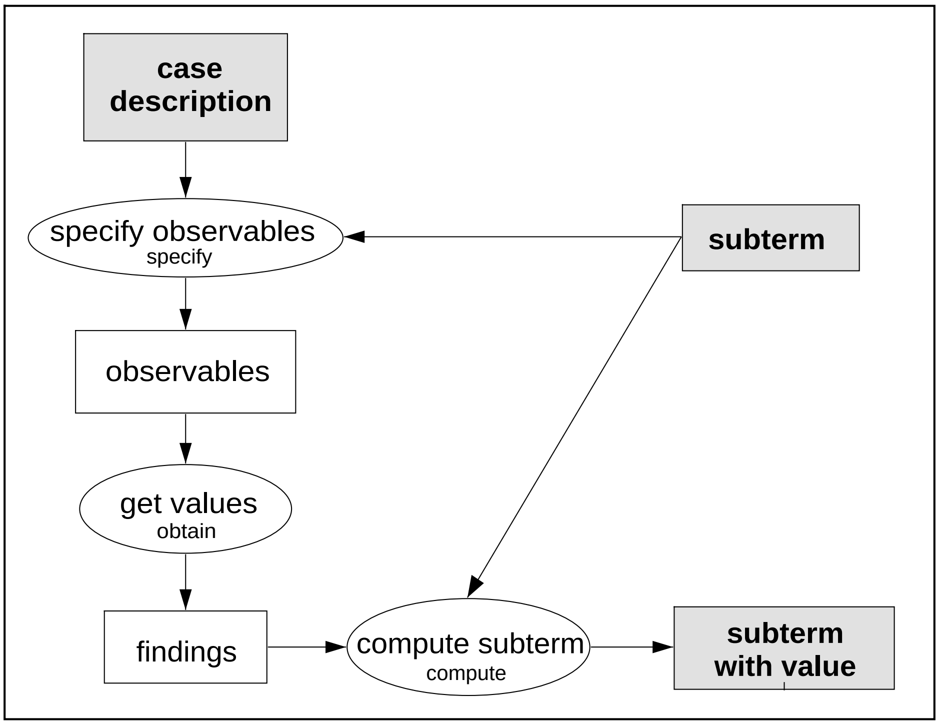



Figure I.5: Inference Structure for *assess subterm*

# References


Aamodt, A. (1991). *A knowledge-intensive approach to problem solving and sustained learning.* PhD thesis, University of Trondheim, Norwegian Institue of Technology, Trondheim, Norway.

Bauer, C. (Ed.). (1992). *Proceedings of 2nd KADS User Meeting,* Munich. SIEMENS A.G.

Brachman, R. J., Fikes, R. E., and Levesque, H. J. (1985). Krypton: A functional approach to knowledge representation. In Levesque, R. J. B. H. J. (Ed.). , *Readings in Knowledge Representation*, 411–429. Morgan Kaufmann.

Bredeweg, B. (1992). *Expertise in Qualitative Prediction of Behaviour.* PhD thesis, University of Amsterdam.

Breuker, J. (1990). *EUROHELP: Developing Intelligent Help Systems.* EC, Amsterdam.

Breuker, J., Winkels, R., and Sandberg, J. (1987a). A shell for intelligent help systems. In *IJCAI-87*, 167–173.

Breuker, J. A., Wielinga, B. J., van Someren, M., de Hoog, R., Schreiber, A. T., de Greef, P., Bredeweg, B., Wielemaker, J., Billault, J. P., Davoodi, M., and Hayward, S. A. (1987b). Model Driven Knowledge Acquisition: Interpretation Models. ESPRIT Project P1098 Deliverable D1 (task A1), University of Amsterdam and STL Ltd.

Brown, D. C. and Chandrasekaran, B. (1989). *Design Problem Solving - Knowledge Structures and Control Strategies.* Research Notes in Artificial Intelligence. Pitman, London.

Chandrasekaran, B. (1986). Generic tasks in knowledge-based reasoning: High-level building blocks for expert system design. *IEEE Expert*, 1(3), 23–30.

Charniak, E. and McDermott, D. (1985). *Introduction to Artificial Intelligence.* Addison-Wesley.

Clancey, W. (1992). Model construction operators. *Artificial Intelligence*, 1(53), 1–115.

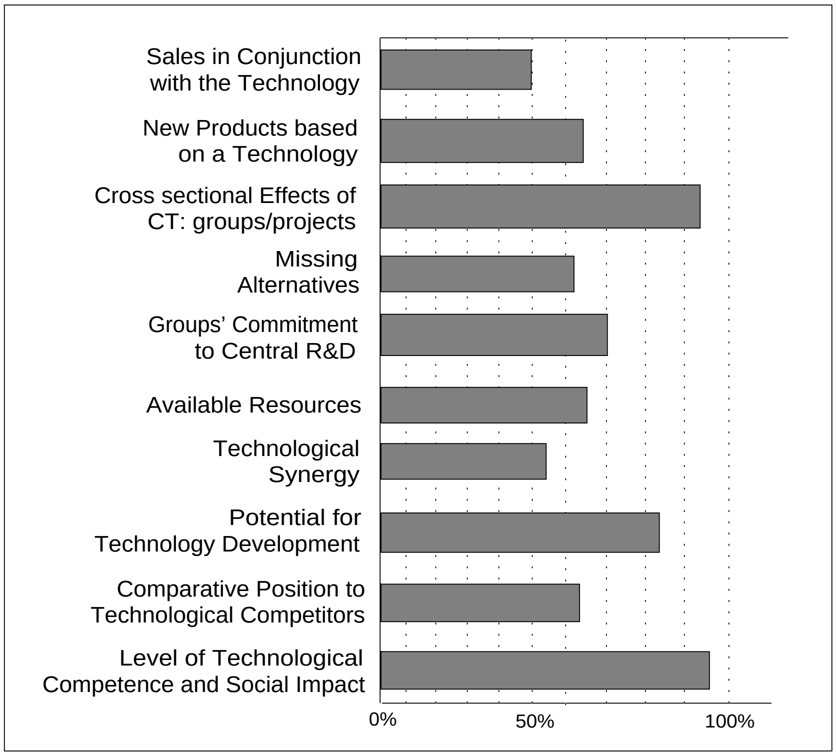



Figure I.6: Criteria Diagram for a (Ficticious) Technology

Clancey, W. J. (1985). Heuristic classification. *Artificial Intelligence*, 27, 289–350.

Clancey, W. J. (1989). The knowledge level reinterpreted: Modeling how systems interact. *Machine Learning, Special Issue on Knowledge Acquisition*, 4(3, 4), 285 – 292.

Costa, M., Frixione, S., Gaglio, G., Spinelli, G., Traversa, M., and Zolezzi, M. (1988). Was king Arthur a king by definition? *AI-Communications*, 1, 32–41.

Descartes, R. (1637). *Discours de la Methode.*

Duursma, C. (1992a). Interpretation models and problem solving methods. In Neumann, B. (Ed.). , *Proceedings of ECAI 92.*, Vienna, Austria.

Duursma, C. (1992b). Role limiting methods for the cm library. Technical Report AI-Memo 92/4, VUB AI lab, Brussels, Belgium.

Eshelman, L. (1988). MOLE: A knowledge-acquisition tool for cover-and-differentiate systems. In Marcus, S. (Ed.). , *Automating Knowledge Acquisition for Expert Systems*, 37–80. Kluwer Academic Publishers, The Netherlands.

Friedland, P. and Iwasaki, Y. (1985). The concept and implementation of skeletal plans. *Journal of Automated Reasoning*, 1, 161–208.

Gasching, J. (1979). *Performance Measurement and Analysis of Certain search Algorithms.* PhD thesis, Dept. of Computer science, Carnegie Mellon University.

Geldof, S. (1991). Kitchen design: Knowledge level analysis. CONSTRUCT Report D/WP2/2, VUB AI-Lab.

Hirst, G. (1989). Ontological assumptions on knowledge representation. In Brachman, R., Levesque, H., and Reiter, R. (Eds.). , *Principles of Knowledge Representation and Reasoning*, 157–169. Morgan

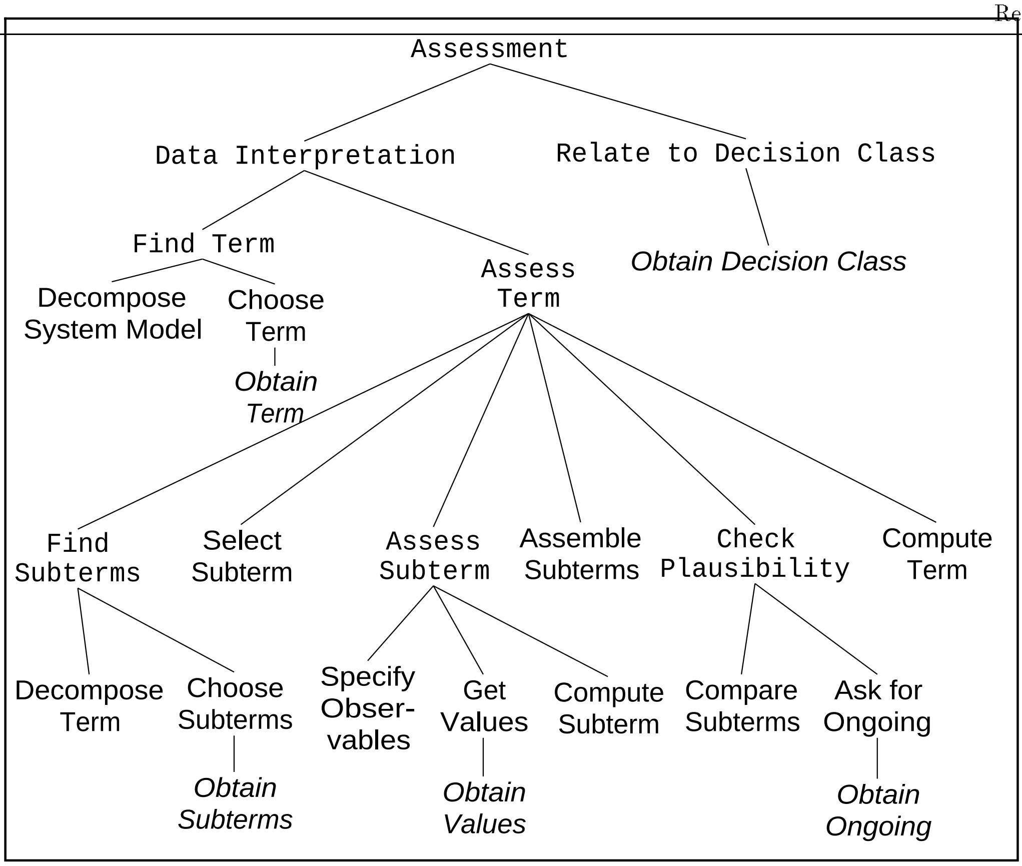



Figure I.7: Decomposition Tree of the Task Assessment


Kaufmann, San Mateo, CA.

Kingston, J. (1992). A risk assessment interpretation model. In Bauer, C. (Ed.). , *Proc. 2nd KADS User Meeting*, Munich, Germany. Siemens.

Knemeyer, U. (1992). Inference structure for the medical assessment in the life insurance underwriting process. In Bauer, C. (Ed.). , *Proc. 2nd KADS User Meeting*, Munich, Germany. Siemens.

Kowalski, R. (1979). *Logic for problem solving*. North-Holland.

Kruger, C. (1990). Interpretatiemodelled voor synthesetaken. Master's thesis, University of Amsterdam, SWI.

Kumar, V. (1992). Algorithms for constraint satisfaction problems. *AI magazine*, 13(1), 32–44.

Laird, J., Rosenbloom, P., and Newell, A. (1986). *Universal subgoaling and chunking, The automatic generation and learning of Goal Hierarchies*. Kluwer Academic Publishers.

Lenat, D. and Guha, R. (1990). *Building large knowledge based systems. Representation and inference in the Cyc project*. Addison-Wesley, Reading Massachusetts.

Lochak, C. (1992). Risk evaluation inference structure. In Bauer, C. (Ed.). , *Proc. 2nd KADS User Meeting*, Munich, Germany. Siemens.

Lockenhoff, C. and Reminger, B. (1992). An inference structure for technology assessment. In Bauer, C. (Ed.). , *Proc. 2nd KADS User Meeting*, Munich, Germany. Siemens.

Marcus, S. (1988). *Automating Knowledge Acquisition for Expert Systems*. Kluwer Academic Publishers.

Martil, R., Wielinga, B., Bright, C., Taylor, R., and deHoog, R. (1992). The commonkads model set. KADS-II Report KADS-II/, KADS-II.

McDermott, J. (1988). Toward a taxonomoy of problem solving methods. In Marcus, S. (Ed.). , *Automating Knowledge Acquisition for Expert Systems*. Kluwer Academic Publishers.

Offutt, D. (1988). A knowledge acquisition tool specialized for the sizing task. In Marcus, S. (Ed.). , *Automating Knowledge Acquisition for Expert Systems*, 175–200. Kluwer Academic Publishers.

Porter, D. (1992). An inference structure for use in fraud assessment domain. In Bauer, C. (Ed.). , *Proc. 2nd KADS User Meeting*, Munich, Germany. Siemens.

Reminger, B. (1990). *Expertensystem zur Unterstutzung der strategischen Technologieplanung*, volume 35 of *Technological Economics Series*. Erich Schmidt Verlag, Berlin.

Reminger, B. (1991). How to manage technology by means of an expert system. In *Proc. of Portland International Conference on Management of Engineering Technology*.

Rich, E. (1983). *Artificial Intelligence*.

Schank, R., Collins, G., and Hunter, L. (1986). Transcending inductive category formation in learning. *Behavorial and Brain Sciences*, 9, 639– 650. Pages 651-686 contains commentaries by several authors.

Schreiber, G. (1992). Bnf for cml. Technical report, Uva.

Steels, L. (1990). Components of expertise. *AI Magazine*, 11(2 (Summer)), 29–49.

Steels, L. (1991a). Commet: A componential methodology for knowledge engineering. CONSTRUCT Report D/WP2/2, VUB AI-Lab.

Steels, L. (1991,b). Keynote speach. Eurovav91, European Workshop on the Verification and Validation of Knowledge Based Systems, Cambridge, England.

Sterling, L. and Shapiro, E. (1986). *The art of Prolog*. MIT Press.

ten Teije, A., van Harmelen, F., and Reinders, M. (1991). Si(ml)2: a prototype interpreter for a subset of (ml)2. KADS-II Report KADS-II/T1.2/TR/UvA/005/1.0, University of Amsterdam.

Van de Velde, W. (1988). *Learning from experience*. PhD thesis, Vrije Universiteit Brussel, Brussel.

van Harmelen, F. and Balder, J. R. (1992). $(ML)^2$: a formal language for KADS models of expertise. *Knowledge Acquisition*, 4(1). Special issue: 'The KADS approach to knowledge engineering'.

van Hees, M. T. (1991a). QUAESTOR, a knowledge based system for computations in preliminary ship design. Private communication.

van Hees, M. T. (1991b). *User Guide -Q-U-A-E-S-T-O-R-/Release 1.00/1991*.

Wielinga, B., de Velde, W. V., Schreiber, G., and Akkermans, H. (1992a). Towards a unification of knowledge modelling approaches. KADS-II Report KADS-II/T1.1/UvA/RR/004/3.0, University of Amsterdam.

Wielinga, B., Schreiber, A., and Breuker, J. (1991). Kads: A modelling approach to knowledge engineering. KADS-II Paper KADS-II/T1.1/PP/UvA/008/1.0, University of Amsterdam.

Wielinga, B. J., Schreiber, A. T., and Breuker, J. A. (1992b). KADS: A modelling approach to knowledge engineering. *Knowledge Acquisition*, 4(1). Special issue 'The KADS approach to knowledge engineering'.

Woods, W. (1975). What's in a link: Foundations for semantic networks. In Bobrow, D. and Collins, A. (Eds.). , *Representation and Understanding: Studies in Cognitive Science*, 35–82. Academic Press, New York, NJ.