Expertise in Qualitative Prediction of Behaviour

Ph.D. thesis (Chapter 3)

University of Amsterdam
Amsterdam, The Netherlands

1992

Bert Bredeweg

Chapter 3

Modelling Problem Solving

This section describes a theory for modelling problem solving behaviour, based on the *KADS* methodology for building Knowledge Based Systems (KBS) [143; 26; 141; 25; 83]. We will use this methodology as a method for integration of the three main approaches to qualitative reasoning described in the previous section. It is therefore relevant that we give a detailed account of the important aspects of this methodology.

The *KADS* methodology is based on a number of principles. Three of these principles are relevant for the research reported in this thesis. The first one is concerned with the distinction between types of basic problem solving tasks:

- *different problem solving tasks can be distinguished by characterising them according to the properties of the input and the expected output.*

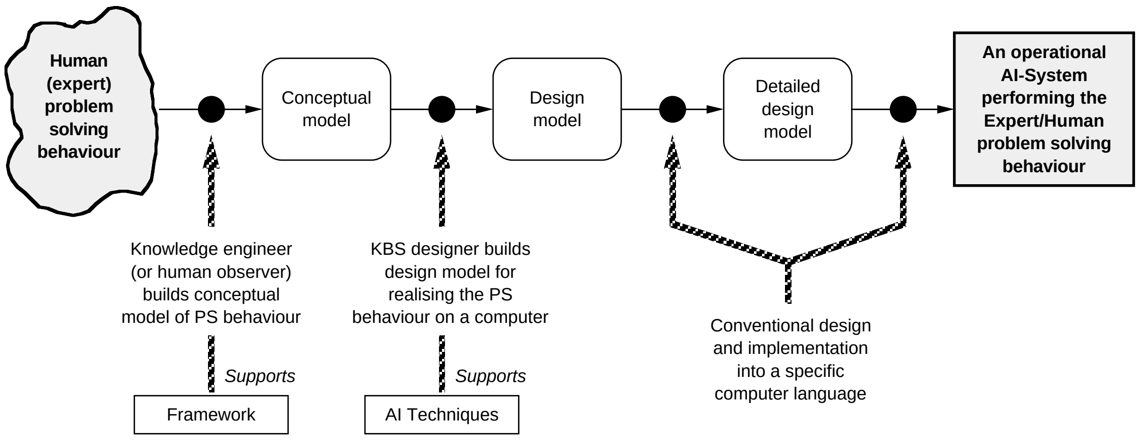
KADS distinguishes between *analysis*, *synthesis* and *modification* tasks. The objective of an analysis task is to determine properties of the system that is object of the reasoning task. Typical analysis tasks are diagnosis, monitoring, and assessment. The objective of a synthesis task is to find a structural description of a system in terms of some given set of elements (vocabulary) or formalism. Typical synthesis tasks are design and planning. The objective of a modification task is to change a certain aspect of the system that is object of the reasoning task. Typical modifications tasks are repair and remedy.

Qualitative prediction of behaviour can be classified as an analysis task during which new properties of the system, namely those that specify its behaviour, are derived from a structural description of the system. In addition, this problem solving task is characterised by the fact that the behaviour descriptions have a qualitative nature, instead of a quantitative nature.

The second principle concerns the notion of using multiple models to bridge the large gap between the problem solving potential of a human expert and realisation of that potential in a computer program (see also figure 3.1):

- *a description of problem solving expertise should distinguish between a description of the problem solving* potential *(independently of the specific implementation) and a description of how the expertise can be realised in a computer program.*

Since Newell [109] introduced the concept of *knowledge level*, several authors [36; 40; 143] have argued that this is the right level for representing the knowledge needed for performing a certain problem solving task. At the knowledge level a description of a reasoning process abstracts from the details of a particular implementation and emphasizes the types of knowledge involved in a reasoning task. In *KADS* this description is referred to as the model of expertise.



Human (expert) problem solving behaviour
Conceptual model
Design model
Detailed design model
An operational AI-System performing the Expert/Human problem solving behaviour
Knowledge engineer (or human observer) builds conceptual model of PS behaviour
KBS designer builds design model for realising the PS behaviour on a computer
Conventional design and implementation into a specific computer language
Supports
Supports
Framework
AI Techniques

Figure 3.1: Intermediate models for knowledge acquisition

The third principle is concerned with how knowledge can be represented in a model of expertise:

- *the knowledge underlying problem solving expertise can be categorised into different types corresponding to different roles the knowledge plays in the reasoning process.*

The model of expertise, and in particular the distinction between different types of knowledge, is further described in section 3.1.

The model of expertise represents knowledge independently from a specific implementation. In order to arrive at an implementation the conceptual model[1] has to be transformed into a design model: a model of the same expertise but orientated towards the artifact that has to be build. A design model should contain sufficient information for a system builder to implement a computer program. The design model is further described in section 3.2.

3.1 Conceptual Model of Problem Solving Expertise

Although terminology is different, a common view appears to emerge based on the idea that different types of knowledge constitute the knowledge level and that these different types of knowledge play different roles in the reasoning process and have inherently different structuring principles [125]. This knowledge typing varies from separation between *declarative domain* and *control* knowledge, as argued for by Clancey [40], to advanced frameworks consisting of multiple layers with complicated internal structures.

KADS distinguishes between four types of knowledge (see figure 3.2). The *domain* knowledge refers to the declarative domain specific knowledge. The *inference* knowledge

[1]There is a distinction between the notion of a *conceptual* model and a model of *expertise* in the *KADS* methodology. The conceptual model further refines a model of expertise with respect to task allocation, i.e. different problem solvers may realise specific parts of the model of expertise (see also 3.2).

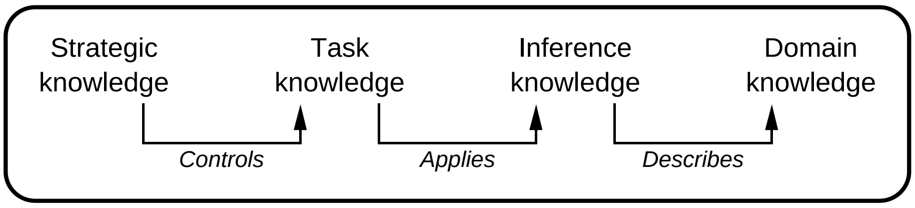


Figure 3.2: Four layered structure for modelling expertise

specifies how the domain knowledge can be used for making inferences. The *task* knowledge describes how the inference potential can be applied for realising problem solving goals. The *strategic* knowledge controls the overall reasoning process.

3.1.1 The Domain Layer

At the domain layer the domain specific knowledge is represented. This knowledge is essentially declarative and task-neutral. In other words, it does not specify what control regime is needed for making inferences, and it is not represented in a specific way to support a certain reasoning task (cf. [140]). The knowledge represented at the domain layer can be used for different problem solving tasks, but the domain layer does not specify how.

3.1.1.1 The Knowledge Representation

For representing domain knowledge *KADS* uses the following modelling primitives: *concepts*, *relations* between concepts, and *structures* built from relations. Concepts refer to anything that a mind, or for that matter a knowledge engineer, can distinguish in the application domain. Concepts have an internal structure that consists of attributes, values, and value restrictions belonging to those attributes (cf. [12]). Typical concepts in the domain of electronics are, for example: transistors, switches, push-buttons, wires, and so forth.

Relations describe dependencies between concepts and form the basis for making inferences. Typical examples are: *causal* relations between causes and complaints, *part-of* relations between a structure and its subparts, and *subsumption*[2] relations between general class descriptions and more specific ones, like, for example, a push-button is a kind of switch. Frequently occurring relations can be classified as follows:

- *Grouping/structural relations*
  These relations describe static dependencies between concepts. Typical examples are *is-a, consists-of, belongs-to* and *spatial* relations.

- *Sequence/time relations*
  These relations describe an ordering in time of events, processes, actions, and other dynamic aspects. Typical examples are *before, after, during, etc.* (cf. [102; 1]).

[2]Alternative names for this relation are *kind-of* and *is-a*.

- *Effect/dependency relations*
  These relations describe causal, associational and functional dependencies. Typical examples are *cause, explain* and *has-function*.

Structures are configurations of related concepts. Whether some entity from the real-world must be represented as a concept or as a structure depends on the amount of detail that is required by the reasoning process. Usually, a structure refers to entities that have a more complex internal organisation than just a set of attribute/value pairs. Consider, for example, an audio amplifier. If the reasoning process requires only knowledge about the weight and the colour of this object, then it can easily be considered a concept. If on the other hand, the amplifier is malfunctioning and therefore subject of a diagnostic reasoning process, then it is more appropriate to regard it as a complex structure that can be decomposed into (many) subparts.

3.1.1.2 Two Points of Discussion

Notice that the knowledge representation, as described above, differs from what in the literature is often referred to as the *domain ontology*. The domain ontology refers to which *type* of concepts, relations and structures are used to represent the domain knowledge present in a specific domain. Describing the behaviour of liquids, for example, may require the notion of *processes* as an ontological primitive (cf. [80]). In the domain layer these processes can be represented either as concepts or as structures.

A second point of discussion is whether it is possible to describe domain knowledge independently from its use (=task neutral). In particular, Chandrasekaran (cf. [36; 27]) opposes this point of view. Rather than trying to separate knowledge from its use, he favours the position that each knowledge representation should integrate a particular way of use (=*interaction hypothesis*).

For the research reported here, the solution to this discussion is not of prime interest, because we are dealing with one specific problem solving task. Moreover, as shown in [19] it turns out that large parts of the framework for qualitative prediction of behaviour (see chapter 4) can be reused directly for model based diagnosis.

3.1.1.3 Possible Extensions

The work of Steels [123] encompasses interesting ideas for enhancing the *KADS* domain knowledge representation. He distinguishes between *domain theory*, *domain model* and *case model* as modelling entities for representing the knowledge of a certain domain. A domain theory is considered to be the principal theory underlying the problem solving in a domain, i.e. describing all the knowledge that is available for such a domain. There are two problems that hamper the use of domain theories in knowledge based systems.

- For a number of domains, for example, most medical domains, not all the relevant knowledge is available and can therefore not be written down as a full domain theory.

- Current artificial intelligence techniques do not allow for representing domain theories to their full extent, because the problem solvers would become too inefficient.

In order to cope with these limitations, a domain model is constructed that represents the relevant aspects of the domain theory in a *sufficient* and *efficient* way. The domain knowledge represents a certain *view* on the domain theory. Some typical examples of domain models according to Steels are given in table 3.1. There is a tradeoff between

Type of domain model	**Description**
Structural model	Various components and their properties.
Behavioural model	The behaviour of a system.
Causal model	The causal relations between components.
Fault model	A hierarchy of possible faults.
Repair model	The relations between faults and possible repairs.

Table 3.1: Examples of domain models

sufficient and efficient. If knowledge is represented in more detail, i.e. closer to the actual domain theory, and therefore better suited for realising different kinds of problem solving, then the problem solving process as a whole is likely to become less efficient.

Domain models cannot be incorporated in *KADS* without reconsidering the interaction hypothesis. Domain models by definition incorporate a certain kind of use and are therefore not independent of how they are used in the problem solving. However, the interesting aspect of domain models is the potential classification of relations they point out. Each domain model is characterised by the specific relation (or set of relations) used to describe the dependencies between the concepts and/or the structures within the model. For example, in the causal model a network of cause-relations between concepts is used. This classification of relations can be used to enhance the knowledge representation of *KADS* with respect to relations.

A second possible extension of *KADS* concerns the problems that a knowledge based system has to solve. *KADS* does not provide a modelling primitive for representing these problems. It also does not provide a mechanism for representing knowledge that is inferred during the problem solving process. The notion of *case models* as introduced by Steels can be used for this purpose. It captures the following aspects of the problem solving process:

- *input,*
  The problem that has to be solved.
- *intermediate knowledge, and*
  The knowledge that is derived during the reasoning process.
- *output.*
  The solution to the problem.

The notion of case models can be further enhanced by using a meta-level point of view (cf. [46; 134]). In such a perspective the domain model can be used for imposing constraints on the input and the output of a problem solving process. In particular, it can be used for specifying:

- what kind of input problems the problem solver can solve, and
- what the requirements are for a sufficient solution.

This meta-level issue will be further discussed in chapter 7.

3.1.2 The Inference Layer

Knowledge at the inference layer concerns the canonical problem solving actions (knowledge sources) that are the basis of reasoning. They represent problem solving competence and are primitive in the sense that other parts of the problem solver cannot influence their internal control.

3.1.2.1 The Knowledge Representation

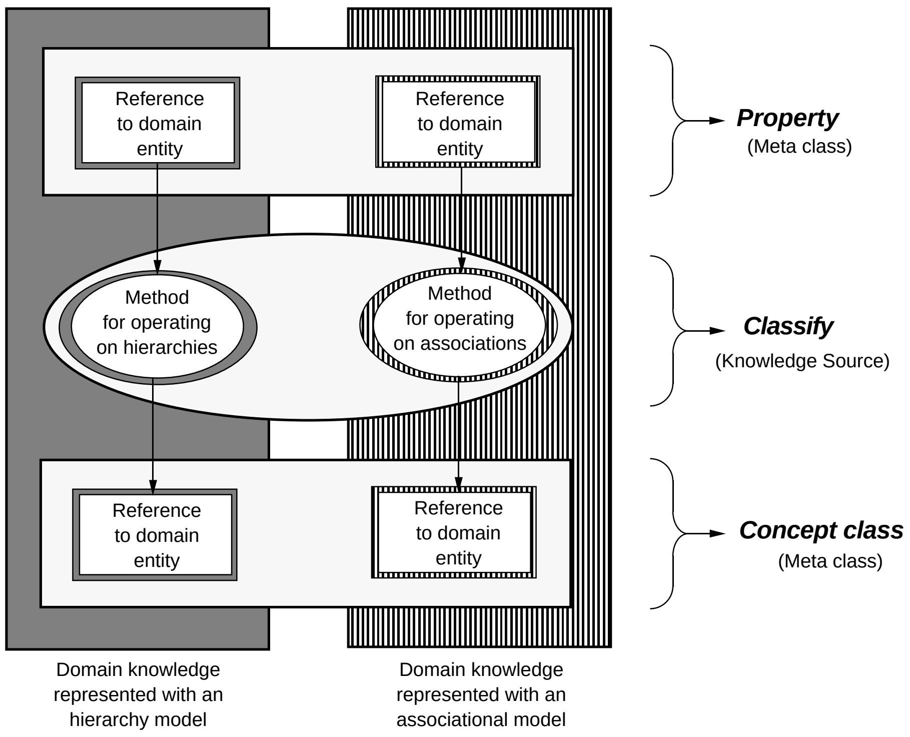
KADS uses two primitives for representing the knowledge at the inference layer: *meta classes* and *knowledge sources*. Meta classes describe the *roles* that the domain concepts and structures play in the reasoning process. In diagnosis, for example, a domain concept such as *(faulty-)transistor* may play the role of a *hypothesis*, or of a *diagnosis*. Each role represents a different, domain independent, use of the domain concept for diagnostic reasoning.

A meta class has no internal structure, it is a slot that can be filled with a domain entity (concept, relation, or structure). It is best understood when regarded as a meaningful placeholder that provides the knowledge sources with references to the domain layer. The roles that can be distinguished are specific to a certain kind of reasoning task, which makes it difficult to come up with a single typology of meta classes. For example the notion of a *solution*, which is essentially similar in all problem solving processes, can be given different, more meaningful names, depending on the specific reasoning task that is going on. In diagnostic reasoning it will be called a *diagnosis*, in design tasks it will be called a *design* and in qualitative reasoning it may be a *graph of behaviours*.

Knowledge sources are functional descriptions of primitive inference making processes. At the domain layer an inference is concerned with applying some operation (*problem solving method*) to concepts, and/or structures, in order to establish the truth value of a certain relation between those concepts and/or structures. The knowledge source is an abstract description of this, it records the competence independently from the domain specific relation that is used by the inference making process. In figure 3.3, for example, two domains are distinguished. In one domain the knowledge is represented as *heuristic associations* between properties and concepts, whereas in the other domain the knowledge is built into a hierarchy of concepts by using *subsumption* relations. Both types of domain relations can be used for the inferencing process needed for implementing the competence required by the knowledge source *classify*.

In addition to different domain relations being used for *one* knowledge source, a specific domain relation can be used for realising the inferencing process that is required by *different* knowledge sources. For example, the knowledge sources *specify* and *abstract* may both use a subsumption relation to make their inference. In the case of specify the subsumption relation is used to go further down a hierarchy of concepts in order to derive more specific attributes. The knowledge source *abstract*,[3] on the other hand, is used to go up the subsumption hierarchy and to find a less specific concept, with less attributes defining its characteristics. In other words different competence requirements can be realised by using the same domain relation in different ways.

[3]Abstract is used in the sense of *making less* detailed. Notice that this is different from abstract in the sense of making an abstract model, for example, a mathematical model of some system in the real-world. This latter notion of abstract is referred to as *modelling* in the research reported here.



Reference to domain entity
Reference to domain entity
Property
(Meta class)
Method for operating on hierarchies
Method for operating on associations
Classify
(Knowledge Source)
Reference to domain entity
Reference to domain entity
Concept class
(Meta class)
Domain knowledge represented with an hierarchy model
Domain knowledge represented with an associational model

Figure 3.3: Classify knowledge source with different domain models

Thus the mapping between knowledge sources and problem solving methods is one of *many-to-many*, as depicted in figure 3.4. The distinction between knowledge sources and problem solving methods is relevant, because it separates the machine specific inference processes, and their internal control mechanism, from reasoning about *how* the available competence can be used for realising a certain problem solving goal. This allows the knowledge at the task layer (see 3.1.3), concerned with applying competence to address problem solving goals, to be represented independently from how the knowledge is modelled at the domain layer.

3.1.2.2 Classification of Knowledge Sources

The classification of knowledge sources is based on the epistemological description of semantic networks (*KL_One*) given by Brachman [11; 12]. Recall, that although specific ontologies for representing some amount of domain knowledge may differ, the knowledge must always be represented as concepts with attributes/values pairs, relations, structures, and instances of these generic classes. As a result the classification of knowledge sources is based on 'what can happen to' these entities, except for relations. Relations are not manipulated by knowledge sources, they are not input to, or output from, knowledge sources. Instead a relation refers to how the input can be mapped onto the output and is used to refer to the specific problem solving method that is applied to implement the knowledge source. As a result, the typology of knowledge sources in *KADS* is based on how the knowledge sources manipulate the knowledge classes constituting the knowledge representation, except for the notion of relations.

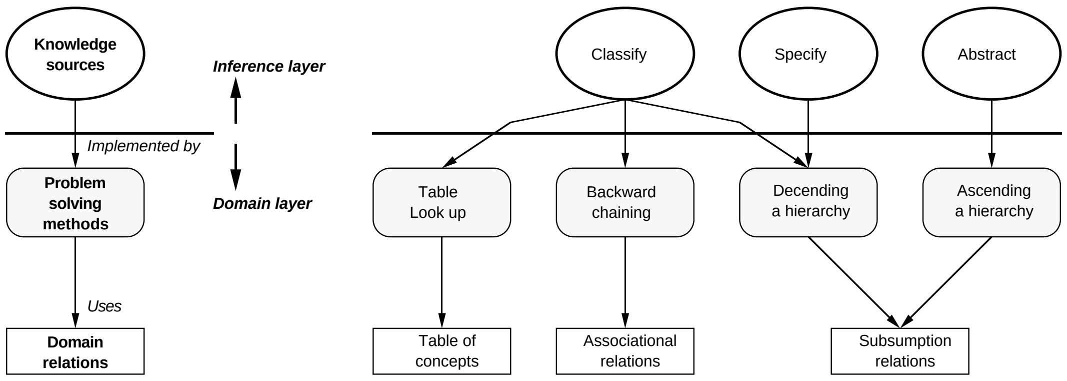


Figure 3.4: Mapping between knowledge sources, methods and relations

The typology of knowledge sources in *KADS* is not an exhaustive one, in the sense that not all the transformations between the knowledge classes in the *KL_One* formalism are considered candidates for knowledge sources. Below we will give a brief summary of the knowledge source typology as defined by *KADS* (cf. [26]).

- **Concept/attribute manipulation**

  Abstract *concept ⇒ concept (with fewer attributes)*
  Abstract is defined as removing an attribute from a concept. The abstract knowledge source makes a concept less specific by abstracting away an irrelevant detail (see also specify). Abstract may also work on an instance.

  Classify *(set of) attribute(s) ⇒ concept*
  The classify knowledge source takes a number of attributes and classifies these as representing a certain concept. In other words, a concept is recognised on the basis of its attributes. Another term for classify could be *identify*. Classify does not distinguish between generic concepts or instances of these concepts, although it usually will have a generic concept as output.

  Generalise *(set of) concept(s) ⇒ (generic) concept*
  Generalisation is concerned with finding the common features in a number of concepts and trying to map these onto an existing concept or to develop a new concept. In the former, generalisation is closely related to classify. Developing a new concept is also known as induction, a reasoning technique often studied in the context of machine learning (cf. [39; 106]). Generalise can also work on instances, but will usually produce a generic concept as output.

  Instantiate *concept (or structure) ⇒ instantiated concept (or structure)*
  This knowledge source creates an instance of a concept or a structure. The inference competence represented by this knowledge source may seem a bit trivial, because it involves some form of copying rather than actually inferring a proper relation. However, the distinction between general classes and specific instances has to be made and that is exactly what this knowledge source does. Notice that this knowledge source is often an implicit part of other knowledge sources.

Specify *concept ⇒ concept (with additional attribute/value pair(s))*
The competence represented by the specify knowledge source is the inverse of abstract. It produces a concept with at least one more attribute/value pair than the input concept. Another term for specification could be *refine*. Assign value (see below) is often an integrated part of a specification inference. Specify can also work on an instance.

- **Attribute/value manipulation (assignment)**

Assign value *(concept) attribute ⇒ (concept) attribute with value*
This knowledge source derives values for attributes of concepts and assigns them to these attributes, for example, by using default mechanisms. Assign may sometimes involve complex inference procedures for inferring the value that must be assigned. In such cases it is better to regard this knowledge source as a *compound* knowledge source and *decompose* it into more primitive inference steps (see also 3.1.3). The assign knowledge source also includes *removing*, *changing* and *overwriting* values.

Compute *structure ⇒ (concept) attribute in structure has value assigned*
Compute is similar to assign except that it takes a structure as input and assigns a value to an attribute of a concept that is part of the structure. The computation of the value is based on the interdependencies between the concepts that constitute the structure. In most cases the structure refers to a formula or an arithmetic operation, which makes compute closely related to arithmetic computation. As a result of this compute is not strictly a primitive knowledge source, but may involve complex inference procedures.

- **Attribute/value manipulation (comparison)**

Compare *value of X, value of Y ⇒ (concept) attribute with difference value*
This knowledge source compares the values of attributes of concepts and produces a concept that represents the difference between those values. Examples of such output are: equal, not equal, or a difference measure between the values.

Match *structure of X, structure of Y ⇒ (concept) attribute with difference value*
Match has the same objective as compare, except that it derives the difference between two structures instead of two concept attributes.

- **Structure manipulation**

Assemble *set of concepts (or partial structures) ⇒ structure*
Assemble takes a set of concepts or partial structures, and creates a structure in which these are configured in such a way that they obey certain design requirements. Another term for this knowledge source could be *compose*. Assemble can also operate on instances.

Decompose *structure ⇒ set of concepts (or partial structures)*
The knowledge source decompose represents the inverse competence of assemble, namely decomposing a structure into the concepts, or partial structures, that constitute the input structure. Decompose can also operate on instances.

Transform *structure ⇒ structure*
This knowledge source transforms one structure into another structure. There are two types of transformation. The first type leaves the input structure as it is and reorders (possibly removes) the elements within the structure (see also the description of sort below). In the second type of transformation a new output structure is created in which the elements from the input structure are assembled. An interesting example of this kind of transformation is the *parsing* of natural language sentences, in which case a linear structure of elements is ordered into a hierarchical structure (often called a parse-tree).[4]

3.1.2.3 Problems with the Typology of Knowledge Sources

The typology of knowledge sources raises a number of problems. The first one concerns the fact that not all the knowledge sources described above can be expressed in the *KL_One* language. In particular, knowledge sources working on *sets* cannot be represented declaratively in this formalism. The typology of knowledge sources extends the formalism by allowing to reason about sets, for example, *generalise* reasons about a set of concepts.

If we want to reason about sets, the typology introduces a second problem, namely that it does not include reasoning about *all* set manipulations. There is, for example, no explicit knowledge source that reasons about merging two sets. For the sake of completeness the following group of knowledge sources can therefore be added.[5] Each knowledge source operates on sets of concepts, attributes, values, structures, or instances.

- **Set manipulation**

Sort *set ⇒ ordered set*
In sort the input elements are ordered into a sequence according to some principle. Sort can be seen as a subtype of both assemble or transform, depending on whether the sorted elements are already in a structure before they are sorted, or not.

Select *set ⇒ one element*
This knowledge source selects, on the basis of some principle, a concept, an attribute, a value, a structure, or an instance, from a set. Select can be seen as a subtype of decompose.

Merge *set Y (or element), set X (or element) ⇒ set*
In merge two sets are merged into a single set, possibly according to some principle. Merge can be seen as a subtype of assemble or transform, depending on whether the sorted elements are already in a structure before they are merged, or not.

[4]As mentioned before, there may be some confusion with respect to the notion of abstract. What in this research is defined as transformation should not be confused with what in the literature is sometimes referred to as abstraction, in particular, in mathematics when researchers talk about building an abstract mathematical model of some system in the real-world. In the research reported here such an activity is called a *modelling* task. Modeling is a basic problem solving task of type design.

[5]The typology as described by *KADS* includes *sort* and *select*, but not *merge*.

It is apparent from the above descriptions that sets can be viewed as certain types of structures. Therefore set manipulations can be described by the knowledge sources that manipulate structures. However, having set manipulations as separate knowledge sources enlarges the expressibility of the knowledge source typology. It is debatable whether set manipulations are actually proper inferences.

The third problem with the typology concerns the label or name given to each knowledge source. Terminology confusion is very likely to occur. An interesting example is the notion of abstract as introduced by Clancey [41]. In his heuristic classification paper he defines three types of *data abstraction*:

- *Definitional abstraction*
  Abstraction based on essential, or necessary, attributes of a concept. For example: if the structure is a one-dimensional network, then its shape is a beam.
- *Qualitative abstraction*
  Abstraction of quantitative data into qualitative definitions, usually with respect to some critical value. For example: if the 'white blood count' in an adult patient is less than 2500, then the 'white blood count' is low.
- *Generalisation*
  Abstraction based on a subtype hierarchy. For example: if someone is a judge, then he is an educated person.

The question to be answered here is to what extent Clancey's abstraction differs from the one given by *KADS*. We can for example try to rewrite the different forms of abstraction as defined by Clancey into the *KADS* framework. Both definitional abstraction and generalisation would then be examples of the specify knowledge source (deriving more attributes of a concept). With respect to the qualitative abstraction it is important to point out that *KADS* uses abstraction for removing irrelevant attributes. Inferring that 'a patient has fever if his temperature is higher than 37 degrees' may be labeled abstraction, because irrelevant detail (the exact value of the temperature) is replaced by the more general notion of fever. However, inferring that 'if a patient has fever, the body temperature must be higher than 37 degrees', may not be labeled abstraction. This type of inference does not remove irrelevant details, or for that matter irrelevant attributes, but instead derives additional attributes and should therefore be labeled as specification.

This brings us to the fourth problem, namely that it may be the case that for certain knowledge representations and/or inference mechanisms it is not possible to rewrite them in terms of the knowledge sources provided by *KADS*. For example, techniques underlying *MOLE* (cf. [65; 64]) are not easily understood in terms of the *KADS* approach. Moreover, the new description of the reasoning competence may lose its expressive power or become less intuitive with respect to the specific characterists of the domain and/or reasoning task. A well understood inference, such as *generating hypotheses* in model based diagnosis (cf. [45; 59; 47; 79]), can be redefined as an *assemble* knowledge source. In particular, in the *GDE* approach [59] this generation consists of assembling components that contributed to the faulty behaviour (discrepancy) into conflict sets (=conflict generation). Each conflict is supposed to contain at least one component that is a possible diagnosis[6] for causing the

[6]Candidate generation is done by assembling components from the conflicts into sets such that the components in these sets account for all the observed discrepancies.

malfunction of the device. Despite the fact that *conflict generation* can easily be rewritten as an 'assemble' knowledge source, the original term is probably better understood by the community of researchers working on diagnosis.

The underlying problem is that in the *KL_One* formalism only a specific set of relations is worked out in detail. These relations belong to what we referred to as the grouping/structural relations (section 3.1.1). These relations are typically suited for representing knowledge about subsumption hierarchies. Other relations, such as time/sequence and effect/dependency relations, are not so easily understood in the *KL_One* formalism. The dependencies that these relations represent are not directly, i.e. declaratively, available in the formalism. It is also not clear if reasoning about these relations actually requires different knowledge sources, possibly using knowledge classes currently not defined in the *KL_One* formalism. The essential question to be answered here is whether such dependencies could be rewritten into the existing ones. It is tempting to argue that such a rewrite is possible, although the dependencies would probably lose some of their intuitive appeal (as explained above for *GDE*), but scientific proof for this can currently not be given.

3.1.2.4 Inference Structure

An inference structure describes the knowledge sources with their input and output dependencies for a certain basic problem solving task. It specifies the problem solving potential of such a basic task. In figure 3.5 an inference structure is presented for the basic problem solving task *monitoring*. This example is taken from [26].

There are two *select* knowledge sources in this inference structure. One of these selects the observables from the real-world (universe of observables), i.e. it determines what *can* be observed in that world. The other one selects parameters from a model, guided by selection criteria. The output of the selection that is done first may be used as additional input for the other. In other words, the monitoring can be guided by a model that prescribes what must be observed, or can be guided by the phenomena of the real-world system. The order in which knowledge sources are used (=control) is not part of the inference layer. The inference structure only describes the knowledge sources with their input and output dependencies (see also 3.1.3).

The *specify* knowledge source determines the norms that are used by the *compare* knowledge source for inferring the difference between these norms and the *obtained* findings. Finally, the *classify* knowledge source determines into what discrepancy class the difference can be classified.

It is important to realise that *obtain* is not a knowledge source (in typical *KADS* inference structures it would be left out). Obtain refers to information that must be gathered from the 'outside-world' and not to an inference step that can be carried out (see also notion of transfer tasks discussed below).

3.1.3 The Task Layer

The knowledge at the task layer is concerned with controlling how the problem solving competence, represented at the inference layer, can be applied for achieving problem solving *goals*.

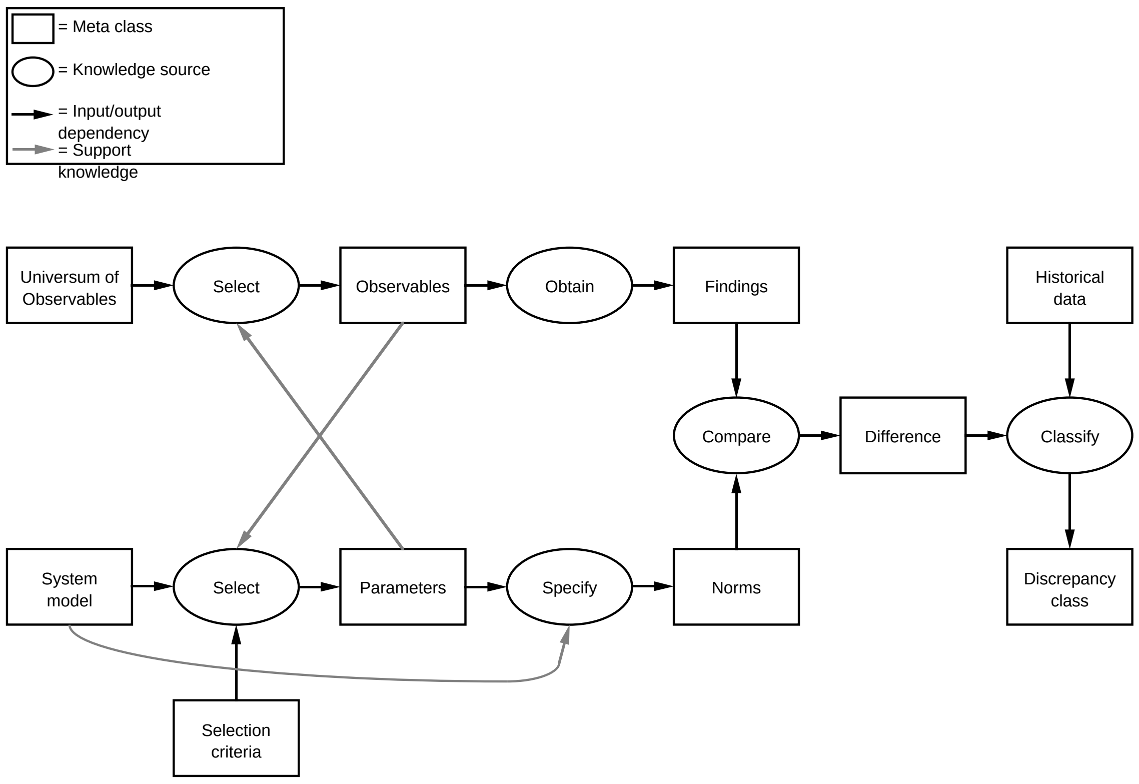


Figure 3.5: Inference structure for monitoring

3.1.3.1 The Knowledge Representation

The knowledge at this layer is represented by *tasks*.[7] Tasks can be decomposed into subtasks. Each task itself is characterised by the specific input that it requires, the output that it generates, and the goal that it can realise. In figure 3.6 a *task decomposition structure* is given for solving physics problems (cf. [104; 105]). This example is taken from Jansweijer [89]. The goal of **solve problem** is to provide a solution to a certain physics problem. This task can be decomposed into **analysing** the problem, **solving** it, and **evaluating** the answer. Three tasks have to be executed to analyse the problem, first **read** it, then make a **sketch** and finally **schematise**. The sketch task refers to making a drawing that represents the problem. This drawing is input for schematise. The purpose of the latter is to arrive at a standard, possibly formalised, description of the problem. This standard description is not only the output of schematise, but also the output of analyse, and consequently the input for **solve**. The solve task consists of two subtasks **assemble** and **compute**. The goal of assemble is to find all the equations that are needed to solve the problem. Such a set of equations is called a mathematical model and is in this case input for compute. Compute has to find the solution to the set of equations. The output of compute is not only the output of solve, but will also contain the answer

[7]The notion of tasks at the task layer should not be confused with the notion of basic tasks as described in the beginning of this chapter. The latter refers to global problem solving tasks, such as, diagnosis, design and monitoring, whereas the former refers to controlling the order in which knowledge sources are applied.

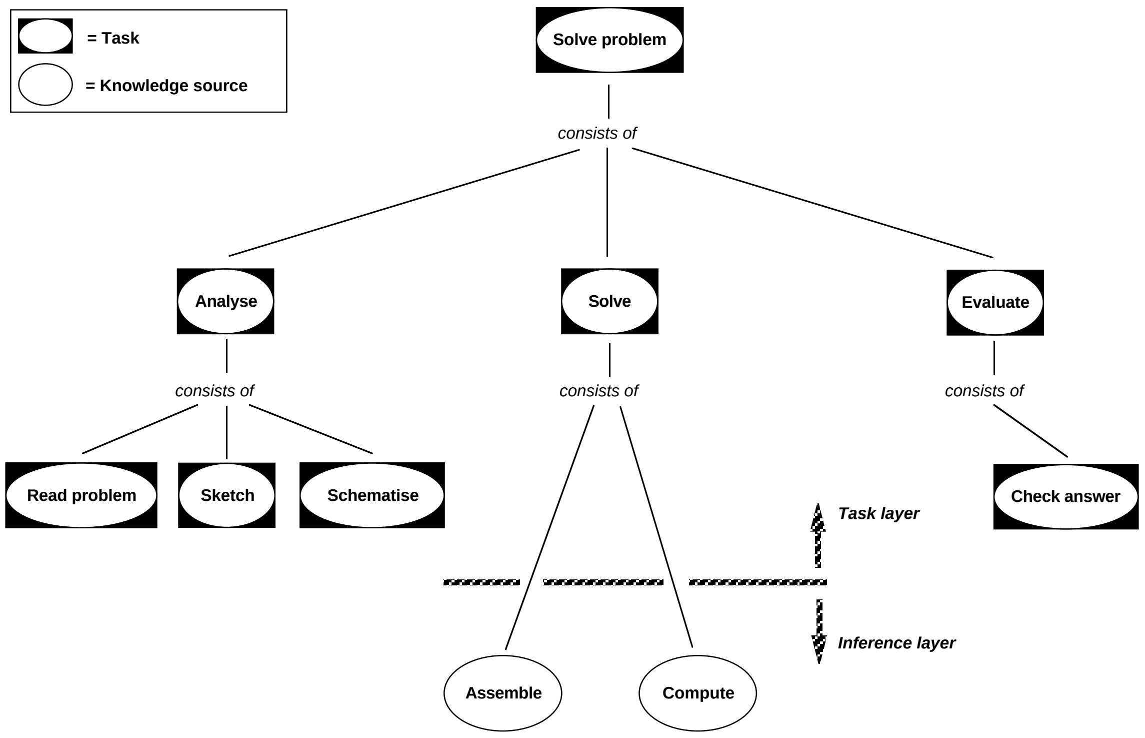


Figure 3.6: Task decomposition structure for physics problem solving

to the initial problem solving task (solve problem). Finally, this answer will be given to **evaluate**, whose goal it is to check and evaluate the outcome. The problem is solved if the evaluation turns out to be positive.

The purpose of decomposing tasks into subtasks is to arrive at sufficiently small subtasks, that each of them can be solved by the inference competence available at the inference layer. This means that the leaves of the task decomposition structure will be knowledge sources. In other words, tasks have to be decomposed into subtasks until the *inference competence* needed to execute the task is small enough to be addressed by a specific knowledge source. In the example above this is the case for the two subtasks of solve, namely assemble and compute. They refer to knowledge sources at the inference layer.

A task decomposition is the result of strategic reasoning at the strategy layer (see below). Tasks themselves have no knowledge about how they can be composed or decomposed into other tasks. In fact, each task generates a number of new subgoals that have to be achieved. The strategic layer decides which tasks can be used to realise these goals. Assigning tasks in this manner results in a specific task decomposition.

In case of a routine problem solving process, the task decomposition always proceeds in the same way. The specific decomposition structure is then the result of what can be defined as a *fixed strategy*.

In *KADS* the notion of tasks has not been given as much attention as the notion of knowledge sources. This is understandable as soon as one realises how complex and large the amount of control knowledge is that needs to be understood before task decomposition structures can be generated on line during problem solving. In particular, if we want to automate these onto a machine. Consequently, the knowledge modelled at the task layer in typical *KADS* models [26] often contains not more than the specific way in which the knowledge sources are ordered to solve the class of problems particular to the problem solving task. Below, a simple example of this is given for the data-driven and model-driven approach to monitoring problems. It is based on the competence represented in the inference structure in figure 3.5.

In the case of data-driven, the *available* data are used to focus the monitoring (see table 3.2), i.e. the monitoring cycle is triggered on the bases of what can be observed. In the case of model-driven, a model is used to direct the monitoring (see table 3.3), i.e. the monitoring cycle is triggered by a model that prescribes what parameters must be observed.

Order	**K. Source**	**Input**	**Output**	**Support**
1	*Select*	Universe of observables	Observables	
2	*Obtain*	Observables	Findings	
3	*Select*	System model	Parameters	Selection criteria Observables
4	*Specify*	Parameters	Norms	System model
5	*Compare*	Findings Norms	Difference	
6	*Classify*	Difference	Discrepancy class	Historical data

Table 3.2: Data-driven monitoring

Order	**K. Source**	**Input**	**Output**	**Support**
1	*Select*	System model	Parameters	Selection criteria
2	*Specify*	Parameters	Norms	System model
3	*Select*	Universe of observables	Observables	Parameters
4	*Obtain*	Observables	Findings	
5	*Compare*	Findings Norms	Difference	
6	*Classify*	Difference	Discrepancy class	Historical data

Table 3.3: Model-driven monitoring

Finally notice that in we can distinguish between *transfer tasks* and *problem solving* tasks. Transfer tasks in general refer to communication and information exchange between two agents. In *KADS* this is part of the *modality* of a problem solver [48]. Modality is also concerned with the task distribution between the artifact and the user. **Obtain data** and **read problem** are examples of transfer tasks. In the research reported here we are primarily concerned with problem solving tasks.

3.1.4 The Strategic Layer

The knowledge represented at the strategic layer is concerned with the overall control of the reasoning process. It can be seen as a kind of meta level [46; 134] that coordinates the activities at the other layers. In particular, it is concerned with how tasks (structures) can be selected and aggregated for realising problem solving goals.

3.1.4.1 The Knowledge Representation

In [143] Wielinga and Breuker present the strategic layer as consisting of a number of basic problem solving tasks. Their approach is depicted in figure 3.7. Given a desired problem

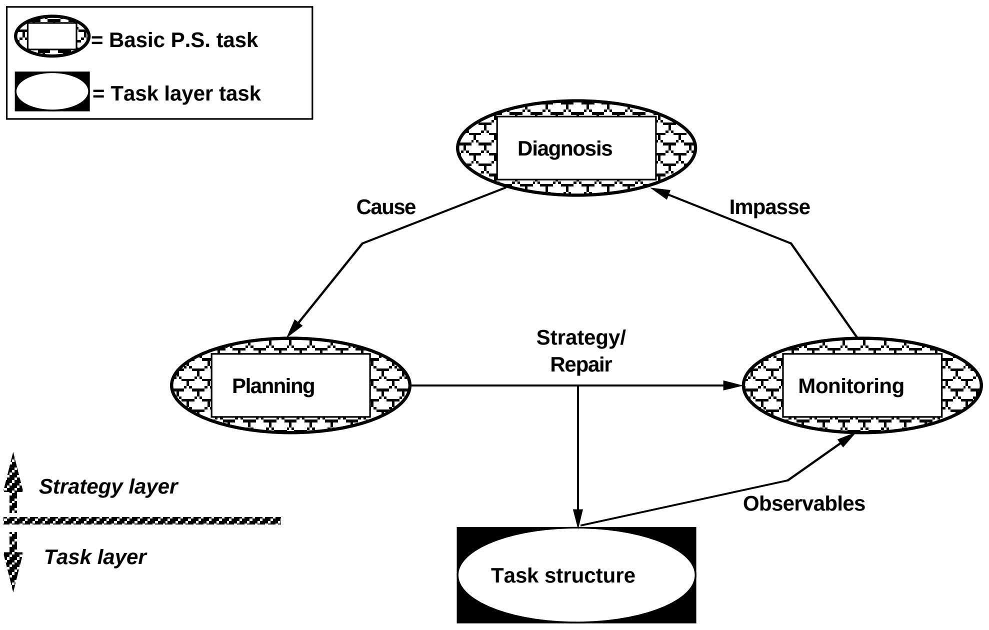


Figure 3.7: Structure of the strategic reasoning process

solving goal, the **planning** task plans a specific strategy that should be able to address the goal. This planning can range from selecting an available task structure from the task layer, to complex configuration of individual subtasks. Once the strategy is present it can be executed. The execution is **monitored** to detect possible deviations from the expected problem solving behaviour. In particular, the monitoring is concerned with recognising impasses that hamper the ongoing of the reasoning. If an impasse is detected it is given to the **diagnoser** which tries to find the cause of the impasse. This cause is input for the planner which can adjust the current strategy such that the impasse is removed and the problem solving can continue. This adjustment may range from a minor modification in the current strategy, and its execution, to entirely replacing it by a new one.

3.1.4.2 Characteristics of Tasks

One of the essential issues that should be modelled at the strategic layer is knowledge *about* problem solving tasks. For example, what goals can be realised and how they can be decomposed into subtasks. Except for the structure described above *KADS* says little about this matter. However, in the components of expertise [123] this is precisely one of the main topics. Steels sees problem solving as a conglomerate of mutually dependent tasks, the *task-structure*, which is the backbone of the problem solving process. In his view, tasks can be analysed from both a *conceptual* and a *pragmatic* point of view. From a conceptual point of view tasks are characterised in terms of the problem that has to be solved. This characterisation is based on the properties of the input and output, and the nature of the operation taking place to facilitate the mapping between them. Examples of tasks are given in table 3.4.

Task type	**Input**	**Output**	**Mapping**
Diagnosis	Observed symptoms	Explanation of how the symptoms came about	Not given
Interpretation (classification)	Observed data	Categorisation of the data	See table 3.5
Design	Specifications	Object that conforms to the specifications	Not given

Table 3.4: Conceptual view on tasks

The mappings are not given for the diagnostic and design tasks, but the idea is that task decomposition methods can be found that decompose these tasks into subtasks until each task is solvable, because there is an inference method that can make the required inference. This means that not all (sub)tasks have the same status: some of them are solvable and some of them have to be decomposed. This matches very nicely with the *KADS* perspective of tasks versus knowledge sources. The classification task mentioned above is in fact a knowledge source, i.e. a leaf of the task decomposition structure, whose inference requirements can be realised by a problem solving method. The other two tasks are too complex and must first be decomposed into subtasks.

The second point of view on tasks, the pragmatic one, focuses, according to Steels, on the constraints in the task that result from the environment in which the system will operate or from the limitations that humans (but also computers) have. He defines three categories of limitations:

- *Limitation in time and space*
  Both the time to reach a decision and the memory for storing knowledge, are always finite.

- *Limitation in observation*
  The necessary data may not be available or may not have the required degree of precision.

- *Limitation in theory formation*
  Models must be inductively derived using real-world interaction or communication with other humans, which often limits the models in their accuracy and scope of prediction.[8]

[8]Another reason why theories can be limited, is simply because they have not yet been fully developed.

Steels then goes on by showing the constraints that may follow from these limitations. He gives the following four examples:

- *Need to avoid search*
  For example: in a diagnostic task the symptoms used as input may have an associated cost, and part of the problem may be to minimise the costs and therefore restrict the number of observations.

- *Deal with weak inference rules or weakly defined concepts (that is, not defined in necessary and sufficient conditions)*
  For example: in a classification task the categories into which the data must be classified may not be strict, that is, they may only be definable in terms of typical features and not in terms of necessary and sufficient conditions.

- *Handle incomplete, inconsistent, and uncertain data*
  For example: in an interpretation task the data to be interpreted may show errors or may be incomplete.[9]

- *Handle explosions of information*
  For example: in a design task the specifications for a design may be incomplete or inconsistent, or the number of possible combinations may be so large that exhaustive search is impossible.

Steels uses tasks in a way similar to problems. He does not discriminate between solving a problem and executing a task. He also does not distinguish between a conceptual description of the problem solving competence and how this competence can be realised in a computer program. In our approach we do distinguish between these notions, which can be used to further discriminate between the limitations mentioned above. Limitations in time and space clearly differ for humans and machines. Therefore the constraints that result from *handle explosions of information* may be different for humans and computers. An interesting example in this respect is an experiment in building knowledge based systems described by Barthélemy [3; 4]. The purpose of the artifact was to design moulds for fabricating plastic parts used in the heating system of a car. One of the main problems was finding the best way to configure as many parts as possible on one mould. As the number of possible configurations was large, the human expert used heuristics to design the moulds. However, it turned out that these heuristics were quite difficult to model into a computer program. Moreover, the computer could easily generate all possible mould configurations, calculate the costs for each configuration,[10] and then select the best design (the fewer moulds needed to create all the parts, the better the design). In other words, different problem solving methods were used by human and machine for dealing with the problem. We consider limitations related to realising competence on a machine part of the design model. This is discussed in section 3.2.

Typical examples are social sciences, such as, psychology and sociology. Steels does not mention this type of limitation in theory formation.

[9]From Steels' work it is not clear how an interpretation task differs from either a diagnosis or a classification task.

[10]In the AI literature this method is known as *generate and test* (cf. [113]).

Distinguishing between problems and tasks is also relevant. For example, dealing with incomplete, inconsistent and uncertain data is a characteristic of the *problem*, not of the task that is used to solve the problem. Any task, be it diagnosis, design, or classification, can in principle be confronted with this constraint. The issue is that each task requires a certain type and status of input in order to be applicable. If this is not the case, for example, because of the limitations mentioned above, then additional tasks are needed to adjust the input accordingly. However, the limitations, and the resulting constraints, are characteristics, or *features*, of the problem, not of the tasks.

In section 3.1.1 we discussed the case model as being the domain knowledge modelling construct for representing input, output and intermediate knowledge. It is interesting to see that dealing with incomplete, inconsistent and uncertain data are constraints that result from limitations that apply to the *input* modelled in the case model (limitation in observation), whereas limitation in theory formation applies to the knowledge present in the domain model.

3.2 Design Model for Problem Solving Expertise

In *KADS* the computer specific aspects of realising problem solving behaviour are dealt with in the design framework [118; 119]. This framework consists of two aspects, the design process and the design model (see figure 3.8). The design process specifies how the

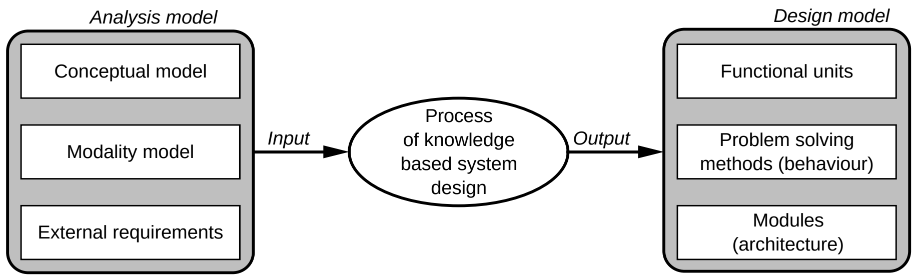


Figure 3.8: The design framework

conceptual model of expertise can be transformed into the different entities of the design model. In particular, it is concerned with the knowledge that can be used to support the construction of the design model. The design model specifies the *functions* that an artifact must be able to address, the machine specific *behaviours* that can be used for realising these functions, and the *architecture* that structures the implementation of the artifact. In the following sections the design process and the different aspects of the design model are described.

3.2.1 Model of Cooperation and External Requirements

Problem solving requires a certain amount of interaction with an environment. In particular, for receiving information about which problems to solve and for presenting the output of the problem solving process. This interaction with the environment imposes additional constraints on the implementation of the artifact. The model of expertise does not cover these requirements. *KADS* therefore distinguishes two additional models that are relevant for designing an artifact:

- a model of cooperation, and
- a model of the external requirements.

The model of cooperation is concerned with interaction between the system and the user and is also known as the *model of modality* [48]. It basically describes the *task distribution* between the artifact and the user. It specifies what problem solving tasks are carried out by the artifact and which ones are left to, or required from, the user. The model of cooperation can be seen as a kind of interface that encapsulates the artifact. For the artifact it specifies what tasks have to be performed by the machine, what input these tasks may use and what output they have to provide. It is in fact only after the task distribution has been decided upon, in the modality analysis, that the model of expertise can be developed adequately.

The second model describes the external requirements [2]. The external requirements are similar to pragmatic constraints, as defined by Steels (section 3.1.4.2), that result from the environment in which a machine operates and from the limitations computers have. An example of this emerges when the system as a whole has to run on a relatively small (personal) computer. In such cases additional constraints on the design of the system clearly result from the limited amount of space available for storing information. An example of an environmental constraint may be that the data, needed for the problem solving process, are stored in a database. The interface with this database forms an additional constraint on the development of the artifact.

3.2.2 The Functional Decomposition

A crucial activity in the design process is to take the three models resulting from the conceptual analysis and *decompose* them into a hierarchy of functional blocks (see figure 3.9). During this decomposition process the external requirements and the model of cooperation are integrated with the model of expertise.

Each functional block represents a distinct functional unit of the final artifact. It can be classified according to its *function type* and the *relation* it has with other functional blocks. Typical relations that can exist between functional units are *input/output*, *consists-of*, and *control* relations. The input/output relations specify the interaction between different functional units. The consists-of relations specify how a certain functional unit can be decomposed into other functional units. The control relations are used for modelling the control dependencies that exist between the functional units, for example, that some function must be realised before another function. Control dependencies are often closely related to input/output relations, because required input for an unit may depend on the output provided by another unit. However, they are not necessarily the same.

Typical functions in the design model are:

- *Problem solving functions*
  Problem solving functions are similar to tasks, and/or knowledge sources, in the model of expertise. However, it is to be expected that the decomposition into functional units, in the design model, differs at some level from the task decomposition,

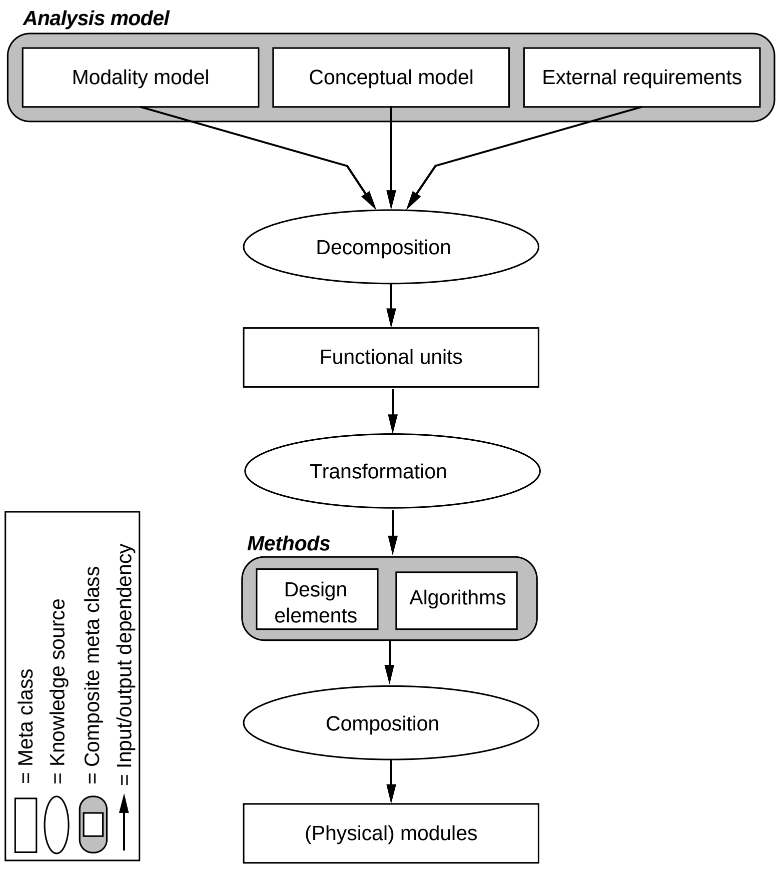


Figure 3.9: Inference structure of the design process

because computer specific aspects are introduced by the modality model and the external requirements.

- *Explanation functions*
  Explanation functions are concerned with explaining answers produced by the artifact. They are typically something that may result from external requirements. The model of expertise, and in particular, the basic problem solving tasks, only describe problem solving competence. It is, however, very likely that for solving a problem different functions are needed for explaining the achieved result. For example, a problem solver based on shallow knowledge may be very good at solving diagnostic problems, but perform poorly if it has to explain why the symptoms are caused by a certain disease, because it lacks the deep knowledge needed to give such an explanation (see section 2.1.4). But also aspects concerned with how the information can be presented in an understandable manner are issues that should be dealt with here.

- *Input and output functions*
  The input/output functions refer to the specific requirements that are imposed upon the artifact. We already discussed the example of the database consultancy. Other examples may be graphical interfaces, line command facilities, or advanced forms of natural language interfaces. Each specific type of input/output requirement puts

additional constraints on the design of the artifact.

- *Information storage functions*
  The storage functions are concerned with storing information (data and knowledge) that is relevant for the problem solving process. These functions are also typical for realising problem solving behaviour on a machine. Specific ways of keeping track of derived results and representing dependencies between intermediate problem solving knowledge, are issues that require computer specific storage functions.

The above set of functions is not necessarily complete, but it illustrates the additional aspects that come into focus once an artifact has to be designed.

3.2.3 Realising the Problem Solving Behaviour

The second crucial activity in the design process is the construction of machine specific behaviours (=problem solving methods) for realising the functions that the artifact must perform. In figure 3.9 this is depicted as a *transformation* step. Problem solving methods realise the mapping between human expert problem solving behaviour and an implementation of that competence in a computer program.

3.2.3.1 Problem Solving Methods: Design Elements and Algorithms

A tentative catalogue of problem solving methods has been provided by Bundy [31]. However, there is little consensus in the artificial intelligence literature on what problem solving methods are. Typical textbooks, as for example [113; 39; 13], only moderately agree on standards. The probably best understood area is that of *state-space* search. Some search methods in this area are depth-first, breadth-first, hill-climbing, best-first, A* (A-star), MiniMax, and others.

In *KADS* a method is defined as consisting of three aspects: a *name*, a set of *design elements* and an *algorithm* (or procedure).[11] The name is just a general label that researchers use to refer to the method. *Hill-climbing* is a typical example of a name. The design elements are the data structures that the method requires for its inference making activity. The algorithm refers to the specific way in which these data structures are manipulated in order to realise the problem solving behaviour.

A simple example of a set of design elements, belonging to the A-star method, is the following (more examples can be found in [118; 119]):

- Goal state
- Begin state
- Current state
- Set of intermediate states
- Measure of costs

[11]In *KADS* (cf. [118]) the algorithm is defined as being one of the design elements. We will use the notion of design elements for referring to data structures only.

    1. Costs for arriving at current goal
    2. Heuristic estimate of (additional) costs for reaching the goal state from the current state

- State operators

The algorithm that manipulates these design elements is well known and relatively simple. It proceeds in steps, starting with the begin state. During each step the operator is applied to the most promising state, expanding it into new states, until one of those states equals the goal state. Determining the most promising state is achieved by adding the estimated cost to reach the goal state to the cost already incurred in arriving at the current state. The intermediate state that has the lowest cost in this respect is the most promising one. An accurate description of the A-star method can be found in [113].

3.2.3.2 Classification Methods Described by Steels

An interesting enumeration of methods is given by Steels [123]. Each of the six methods he describes realises machine specific behaviour required by the knowledge source classify (see section 3.1.2.2), i.e. recognising a certain concept on the basis of a set of attributes. However, all methods have their specific set of design elements and corresponding algorithms. The methods are given in table 3.5.

The differentiation method, as described in the above list, does not realise the competence of the classify knowledge source. It does not classify a set of attributes into a concept, or class. Instead, it tries to find the most differentiating attributes between concepts.

3.2.3.3 Relations between Knowledge Sources and Problem Solving Methods

A question may emerge concerning the difference between the modelling entities in the design model and the conceptual model, i.e. the difference between the building blocks of the methods (design elements and algorithms) and the modelling entities of the inference layer (meta classes and knowledge sources). Knowledge sources differ from algorithms in the sense that they are declarative descriptions that represent dependencies between epistemological classes grounded in the *KL_One* formalism, whereas an algorithm is a procedure that describes a certain set of operations on data structures. In the case of a specific mapping between the conceptual model and the design model, namely that each knowledge source maps onto a single method, the algorithm will manipulate the design elements such that, as a whole, the method implements the competence represented by the knowledge source. It may however, also be the case that a method realises the competence required by more than one knowledge source, or for that matter by a complete task structure. Aspects of transforming the conceptual model into the design model are further discussed in section 3.2.6.

The difference between design elements and meta classes is more complex. Recall that meta classes are place holders that relate domain knowledge to problem solving roles (used in human expert problem solving behaviour) whereas design elements are computer specific data structures that must be available to implement a certain algorithm. Steels argues, along similar lines as McDermott [103], that a problem-solving method has a set of roles which must be filled by specific domain knowledge. For example, weighted

Method name	Brief description of the method
Linear search	For this simple method each concept has to be represented as a set of attributes. The method goes through the possible concepts each time. The concept for which all features match is selected.
Topdown refinement association	This method works on a hierarchy of concepts, with the most general concept being at the top node. While going down the hierarchy more specific attributes are required to hold for the concept. The method systematically searches the hierarchy (starting at the top) and establishes the most appropriate concept at a particular level
Association	This method uses associational relations between an attribute and a concept. It assumes that the domain model indicates which attributes are strongly associated with a concept.
Differentiation	If only a limited set of concepts remains, the concepts can be compared and their differentiating attributes can be computed. These differentiating attributes are good candidates for additional data gathering, so that discrimination between concepts can take place.
Weighted evidence combination	Each attribute (or combination of attributes) may be more or less essential to a concept. In weighted evidence combination each attribute is given a weight that represents the relevance of the attribute to the concept. The best matching concept is the one with the highest score on the sum of the weights. This method is used in Mycin [30], for example, to select the organism causing the infection.
Distance computation	This method calculates the difference between the attributes that must be classified and each of the target concepts. The target concept with the lowest value on this distance metric is the most appropriate concept.

Table 3.5: Classification methods given by Steels

evidence combination requires that we can elicit weights for representing the importance of attributes. In the design model of *KADS* this set of roles is referred to as the design elements of the method. Thus, in the example of weighted evidence combination, the design elements directly map into meta classes in the model of expertise.[12]

This mapping is not always that straightforward. Sometimes the mapping between the problem solving method and the competence required by the conceptual model, is not a simple 'one to one' relation, but requires a complex transformation process. In addition, methods may introduce machine specific requirements that have to be dealt with. For example, in a typical search method one has to construct the algorithm such that the method does not get stuck in a loop (see also section 3.2.6). Steels, and also McDermott, do not explicitly distinguish between human expert problem solving behaviour versus computer specific aspects of realising such behaviour, which confuses the issue.

[12]An underlying problem here is that there is a distinction between problem solving methods that humans use and problem solving methods that realise competence in a computer program. In the research reported here we reserve the term problem solving method for computational methods that realise problem solving behaviour in an artifact.

3.2.3.4 Visualising the Transformation Activity

As mentioned before, the crucial activity in the design process, with respect to realising the machine specific problem solving behaviour, is concerned with *transforming* the functional blocks into design elements and corresponding algorithms. This activity is visualised in figure 3.10. *KADS* assumes that functions can be decomposed into subfunctions until

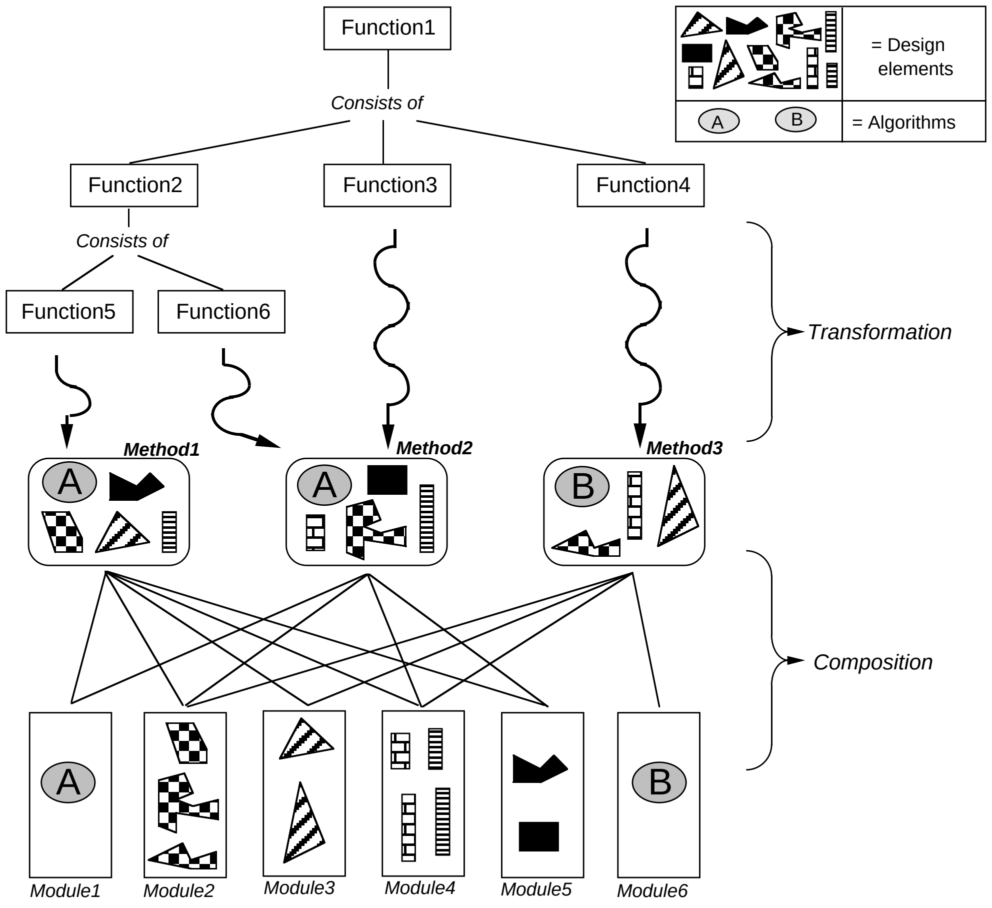


Figure 3.10: Visualising function transformation and module composition

each leaf of the functional decomposition can be implemented by a specific method.

3.2.4 Construction of the Architecture

The third activity in the design process concerns the construction of the architecture of the artifact. This design activity composes the design elements and algorithms, that result from the transformation activity, into coherent modules that will constitute the artifact (see also figure 3.9). Two principles support this activity:

- prevention of knowledge redundancy, and
- minimising coupling and maximising coherence [148].

Prevention of knowledge redundancy dictates that the actual implementation of the artifact should not contain duplications of data structures and inference procedures. If, for example, some method is used more than once, then it makes sense to write one procedure that can be used by each of the methods. Minimising coupling refers to distinguishing between modules in such a way that the interaction between modules is as small as possible. Maximising coherence simply refers to keeping aspects that are similar as much as possible in one module. The idea here is that modifying and debugging the artifact is less complex, when modules are internally coherent and have limited interaction with other modules. The details of this composition activity are visualised in figure 3.10.

KADS introduces the notion of *environments*. An environment is an off-the-shelf software system, that incorporates the design elements and algorithm(s) of one or more problem solving methods. Typical examples are EMYCIN and KEE. Each of these software systems is prefabricated to support the problem solving behaviour as it is facilitated by some specific method.

Interesting research in this direction can be found in the Generic Task (GT) approach (cf. [35; 36; 34; 37]). The generic tasks are associated with off-the-shelf software systems that incorporate a number of methods. Each of those configurations is supposed to be suited to implement a certain reasoning task (see also section 3.2.6).

3.2.5 Going Through the Design Process

Although the description of the design framework discusses the activities in a certain order, there is no prescription in *KADS* that this is also the order in which the design activities must be carried out. Figure 3.9 typically depicts the competence that is required to create a design model and does not specify the control that can be exerted upon these activities. Consequently, the design process may start with each of the three activities. For example, by doing the composition step first, which means that the designer of the artifact decides beforehand upon using a particular software environment for building the system. Thus, before it is known whether that software environment indeed provides the design elements and inference algorithms that are needed for implementing the competence that is required by the conceptual model. The design process may also start with the transformation activity. The designer would then begin by selecting problem solving methods that can be used for realising problem solving competence.

It is to be expected that the design process turns out to be problematic when the designer begins with either the composition or the transformation activity. It is very likely that the entities provided by the shell and/or the methods, provide insufficient functionality for realising the required competence. We therefore favour a design process that starts with the functional decomposition, then proceeds with the transformation into methods, and finally completes the composition of modules. In this approach the required problem solving competence, as described in the analysis model, is fully taken into account during the search for the machine specific realisation of it.

3.2.6 Mapping between Analysis Models and Design Models

KADS distinguishes three forms of how the analysis model can be mapped onto the design model: isomorphic mapping, non-isomorphic mapping, and semi-isomorphic mapping.

These differ with respect to how the modelling entities in the four layer model are mapped onto the functions and methods in the design model. In the case of isomorphic-mapping each entity in the conceptual model maps onto a specific entity in the design model. In case of semi-isomorphic mapping each *type* of modelling entity in the conceptual model maps onto a specific *type* of modelling entity in the design model. This kind of mapping differs from the isomorphic version in the sense that the latter has no type matching between entities from the different models. Finally, in the case of non-isomorphic mapping there is no specific relation between the entities in the conceptual model and entities in the design model.

Studying this classification of mappings in more detail highlights three problematic aspects. Firstly, the modules in the design model are not used in this classification. Secondly, the two additional models that constitute the analysis model (the external requirements and the model of modality) are not taken into account. Thirdly, the identification of isomorphic mapping does not give much insight in the exact nature of the relation between the conceptual model and the design model. Instead it only specifies that each entity in the conceptual model maps onto some entity in the design model. This is similar to non-isomorphic mapping, in the sense that there is no *specific* relation between entities in the conceptual and in the design model.[13]

More useful distinctions between the different forms of transforming the analysis model into the design model are the following:

- Type oriented mapping,
- Task oriented mapping, and
- Unconstrained mapping.

In the case of type oriented mapping each *type* of entity in the analysis model is mapped onto a specific *type* of entity in the design model. This approach, which is illustrated in figure 3.11, is similar to how *KADS* defines semi-isomorphic mapping, i.e. the relation between the two models is defined in terms of how the entity types in both models are related.

In the case of task oriented mapping, the problem solving task in the conceptual model is taken as a starting point. Recall that such a task is decomposed into subtasks until each leaf of the task decomposition bottoms out into a knowledge source. This knowledge source refers to some inference making action that realises the required competence. In task oriented mapping all the knowledge entities that contribute to some problem solving task (= a task at the task layer, or a basic task at the strategic layer) map onto a specific configuration of functions, possibly with the problem solving methods that realise these functions in the design model. In other words, vertical slices in the conceptual model containing related aspects from all three or four layers in the conceptual model, map onto vertical slices in the design model, the latter being agglomerates of functions and, possibly, related methods. Notice that the classification of possible mappings as given by *KADS* does not include this task oriented option. However, this mapping is relevant because it clarifies how other approaches to modelling and realising problem solving can be related

[13]In the case of non-isomorphic mapping two entities in the analysis model may map onto one entity in the design model, which is not allowed in the case of isomorphic mapping.

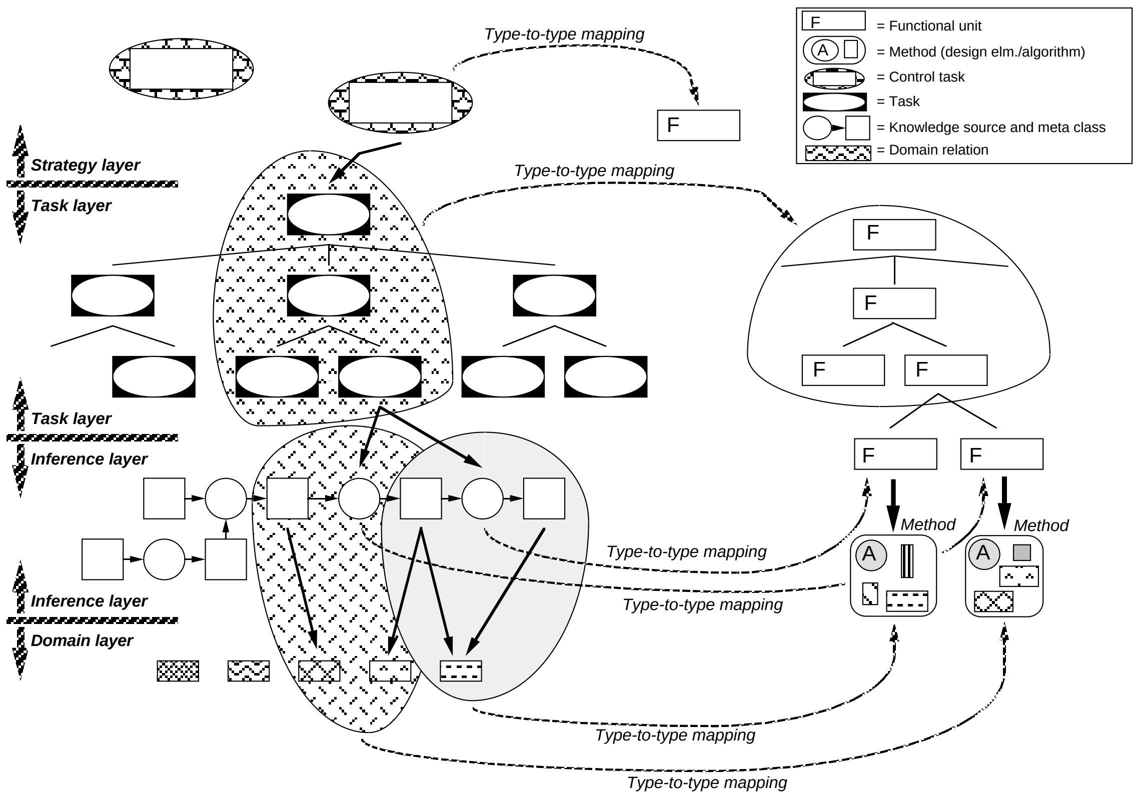


Figure 3.11: Type oriented mapping

to the *KADS* approach. In particular, the GT approach put forward by Chandrasekaran. As mentioned previously, this approach tries to identify off-the-shelf software that can be used for realising problem solving competence required by a generic task. Such an off-the-shelf piece of software is an implementation of machine specific behaviour that is needed to address such competence. The identification process is therefore a task oriented mapping approach to realising human expert problem solving behaviour on a computer. The task oriented mapping is depicted in figure 3.12.

Finally, the unconstrained mapping does not define any specific relation between the entities in the analysis model and those in the design model. Both isomorphic and non-isomorphic mapping, as defined by *KADS*, fall into this category.

3.3 Concluding Remarks

We have presented an overview of the *KADS* methodology for building knowledge based systems and extended it with an interpretation that at some points deviates from other overviews of the methodology (cf. [144]). In addition, we have pointed out places were the methodology can be enhanced as well as issues that are currently unresolved.

The *KADS* approach has proven to be of great use for analysis and interpretation of verbal data provided by human experts (cf. [145; 142; 23; 24]) and the construction of knowledge based systems for the related domain of expertise (cf. [118; 116; 90; 133; 110]).

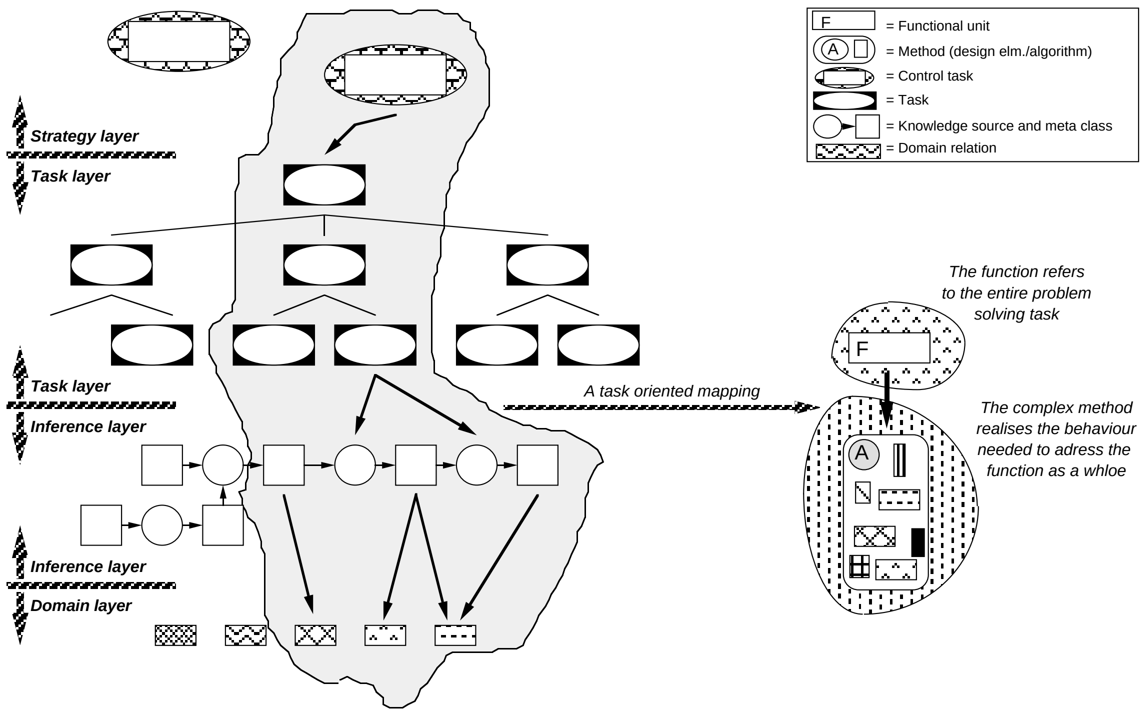

F = Functional unit
A = Method (design elm./algorithm)
= Control task
= Task
= Knowledge source and meta class
= Domain relation
Strategy layer
Task layer
Task layer
Inference layer
Inference layer
Domain layer
A task oriented mapping
The function refers to the entire problem solving task
F
The complex method realises the behaviour needed to adress the function as a whloe
A

Figure 3.12: Task oriented mapping

However, in our research we will use the methodology for a different purpose. Instead of interpreting verbal data provided by human experts, we will investigate existing computer programs in order to develop a conceptual model of the competence that is implemented in these artifacts. By abstracting from the implementation details we will construct a framework for qualitative prediction of behaviour that can be used for comparing the different approaches. In addition, this framework is used for implementing a computer program that has a broader problem solving functionality than each of the original artifacts.