

$(ML)^2$: A formal language for KADS models of expertise*

Frank van Harmelen[†] John Balder[‡]

[†]Department of Social Science Informatics
University of Amsterdam
Roetersstraat 15, 1018 WB Amsterdam, The Netherlands
e-mail: frankh@swi.psy.uva.nl

[‡]Software Engineering & Research Department
Netherlands Energy Research Foundation ECN
P.O. Box 1, NL-1755 ZG Petten (NH), The Netherlands
e-mail: balder@ecn.nl

Abstract

This paper reports on an investigation into a formal language for specifying KADS models of expertise. After arguing the need for and the use of such formal representations, we discuss each of the layers of a KADS model of expertise in the subsequent sections, and define the formal constructions that we use to represent the KADS entities at every layer: order-sorted logic at the domain layer, meta-logic at the inference layer, and dynamic-logic at the task layer. All these constructions together make up $(ML)^2$, the language that we use to represent models of expertise. We illustrate the use of $(ML)^2$ in a small example model. We conclude by describing our experience to date with constructing such formal models in $(ML)^2$, and by discussing some open problems that remain for future work.

1 Introduction

One of the central concerns of “knowledge engineering” is the construction of a model of some problem solving behaviour. This model should eventually lead to the construction of a system that exhibits the same behavior. One of the prominent approaches in recent years to this problem (at

*The research reported here was carried out in the course of the KADS-II and REFLECT project. These projects are partially funded by the ESPRIT and BRA Programmes of the Commission of the European Communities as project numbers 5248 and 3178. The partners in the KADS-II project are Cap Gemini Innovation (F), Cap Gemini Logic (S), Netherlands Energy Research Foundation ECN (NL), ENTEL SA (ESP), 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). The partners in the REFLECT project are the University of Amsterdam (NL), the Netherlands Energy Research Foundation ECN (NL), the National German Research Centre for Computer Science GMD (D) and BSR Consulting (D). This paper reflects the opinions of the authors and not necessarily those of the consortium.

least in Europe) has been the KADS¹ methodology for knowledge engineering (Wielinga, Schreiber & Breuker, 1992). KADS is centered around a so-called *model of expertise* which describes the problem solving expertise of the system to be modelled² independent of a possible implementation.

In many fields of engineering, it is a good practice to formulate models in languages that are as precisely defined as possible. The advantages that accrue from such a formalisation of models (be it models of bridges, machines, electronic devices or problem solving behavior) are the removal of ambiguity, the facilitation of communication, and the possibility of deriving from the model properties of the artifact that is modelled. If “knowledge engineering” wants to live up to its name, and is to become a proper engineering activity, a similar development of the field towards a formal treatment of the models it is concerned with must take place.

In this paper, we will investigate the use of (ML)², a formal language that we have developed for the representation of KADS models of expertise. In another paper (Akkermans, van Harmelen, Schreiber & Wielinga, 1990) we already have introduced some of the elements of (ML)². This paper extends and modifies that previous publication, by making the definitions for the inference layer more specific, and by giving an substantially different formal account of the task-layer. This paper also contains guidelines on how to map KADS notions onto (ML)² constructions. Before we proceed with the presentation of the language, we shall argue how the advantages of formalisation of models, as known from other engineering disciplines, apply in equal measure to the field of knowledge engineering:

Remove ambiguity Traditionally, models of expertise in KADS have been expressed in an informal language consisting of a mixture of structured natural language, diagrams and KL-ONE like representation languages (e.g. (Breuker, Wielinga, van Someren, de Hoog, Schreiber, de Greef, Bredeweg, Wielemaker, Billault & Hayward, 1987)). This has inevitably led to imprecision in the definitions of the components of a model of expertise. A case in point is the definition of *knowledge sources*. These are typically identified by a single term, that is supposed to be indicative of the action that the knowledge source performs. Such a single term plus the accompanying natural language description is often not enough to uniquely define the action of a knowledge source. This has led to a number of different interpretations of knowledge sources to be used in different models of expertise. A formal description of the inference performed by a knowledge source will remove this ambiguity.

Notice that the use of a formal language will not settle any discussions on what a component of a model of expertise (like a knowledge source) *should* do. The use of a formal language is *not normative*, but only *descriptive*. Once it is possible to write down all the different interpretations of a knowledge source in a formal language, external arguments must be used to settle which of these is the most appropriate version.

Facilitate communication An advantage closely related to the removal of ambiguity is the ease of communication that is gained by formalising models of expertise. Models of expertise are a major means of communication in various stages of the development process of a system, both between the expert and the knowledge engineer, and between designers and implementers of a system. However, the informal description of models of expertise is often a barrier to this communication because of its inherent ambiguity, and a more precise formulation of these models of expertise will disambiguate the communication. At the same time, however, the formal models

¹We will not devote any space to describing the KADS methodology. See other contributions to this issue.

²This model-based approach to knowledge engineering is shared among a number of competing approaches, notably Generic Tasks (Chandrasekaran, 1987), Method-to-Task (Musen, 1989), and Role-Limiting-Method (McDermott, 1989). See (Karbach, Voß, Schukey and Drouwen, 1990) for a useful comparison of all these approaches.

will be less accessible to domain experts because of their technical complexity, and can therefore not entirely replace the informal model of expertise.

Bridge gap to design model As correctly stated in (Karbach *et al.*, 1990), a substantial gap remains between a KADS model of expertise and an implementation of this model as an efficient program. The development of a formal version of a model of expertise will help in bridging this gap: although the formal language that we propose is entirely declarative, and is not concerned with making any design decisions relating to algorithms, efficiency, interaction etc., the increased precision of a formal model will make such decisions easier. The use of formal models to verify models of expertise is briefly discussed in section 5.3.

Derivation of properties A formal representation of a model of expertise will make it possible in principle to derive properties of such a model. It is of obvious interest to derive properties like *consistency* (can a contradiction be derived in the formal model), *completeness* (can every conclusion that we expect to derive indeed be derived in the formal model), and *soundness* (is every conclusion that can be derived in the formal model a desired conclusion?). A special case of this use of formal models is as a data-structure for another program that tries to establish properties of the model. This approach is the subject of the REFLECT project (Reinders, Vinkhuyzen, Voß, Akkermans, Balder, Bartsch-Spörl, Bredeweg, Drouwen, van Harmelen, Karbach, Karszen, Schreiber & Wielinga 1991).

We want to stress the point that our formal models are of a purely specificational nature. Unlike other approaches, such as for instance MODEL-K (Karbach *et al.*, 1990), we do not aim at the full mechanization of these models. Formal models are more precise than models of expertise but still they are expressed at a high level of abstraction. They do not contain knowledge concerned with computational efficiency as this is not part of models of expertise in the KADS methodology. Thus, although it is possible to animate (part of) the inference process by interpreting formal models by machine, this is certainly no substitute for the system that is (partly) specified by a formal model.

1.1 Formal versus informal models

Our emphasis on formal models in this paper should not be taken as a desire to completely replace the informal models that have until now been used. We believe that there remains a significant role for informal models, especially in the early stages of the knowledge acquisition process, when many of the concepts and definitions are still ill-formed, and when communication with the domain experts is of crucial importance.

1.2 The use of logic

We have chosen logic as the foundation for our formal specification language (ML)². This is a rather obvious choice, because of logic's long history and established position as a tool for formal analysis. We will not state the case for the use of logic here, since it has been made adequately before (Hayes, 1977; Moore, 1982; Moore, 1984). We only remind the reader of advantages such as a formal semantics, the possibility for stating incomplete information (disjunction, existential quantification), and well-understood properties such as completeness, soundness and (semi)-decidability. Moore (1984) distinguishes three levels at which logic can be used, namely as an analytic tool, as a representational mechanism, and as a computational mechanism. We are using logic only at the first two of these levels, and make no claim about logic as a computational mechanism (unlike for instance (Kowalski, 1979)).

1.3 The structure of this paper

Having motivated the formalisation of models of expertise, we will now present the constructions that we propose our formal representations. In the following sections, we will discuss each of the layers of a KADS model of expertise³, and define the formal constructions that we use to represent the KADS entities at every layer: order-sorted logic at the domain layer (section 2), meta-logic at the inference layer (section 3), and dynamic-logic at the task layer (section 4). All these constructions together make up $(ML)^2$, the language that we use to represent models of expertise. We will illustrate the use of $(ML)^2$ in a small example model. Language elements of $(ML)^2$ are illustrated through this example soon after they are introduced. We conclude by describing our experience to date with constructing such formal models in $(ML)^2$ (section 5), and by discussing some open problems that remain for future work (section 6).

2 Domain layer

The domain layer of a KADS model of expertise is supposed to represent declarative knowledge about the domain of application: facts and rules that are true in the domain, represented independently from how this knowledge is going to be used. The language of logic has been developed to represent exactly this kind of information, and it is therefore not surprising that we chose logic as the representation language for the domain layer (section 2.1). We will extend this language in a conservative way by introducing sub-theories (section 2.2). In section 2.3 we will show how this language can be employed to model KADS notions at the domain layer.

2.1 The language

We use an extended first order predicate calculus with two conservative extensions. One extension concerns the modularization of the knowledge, and will be discussed in the next section. The second extension concerns the syntax of the language itself: We propose to use an *order-sorted language* (for an overview see, *e.g.*, (Schmidt-Schauß, 1989). This is a language in which all variables and constants have associated sorts (types), with these types organised in a subsort hierarchy. Predicates and functions are also typed (according to the types of their arguments, and, for functions only, their results). The inference rules of the logic take into account that two terms can only match if there are appropriate subsort relationships between their sorts. Previous work in AI (Walther, 1984); Cohn, 1985) has shown these languages to have a number of advantages. These include the reduction of the number of axioms, the reduction of the length of logical formulae, a reduced search space during deduction, and increased human readability. Furthermore, sorts provide a natural form for representing certain ontological primitives, as discussed in section 2.3. Such an order-sorted first-order language is a *conservative extension* of a normal first-order language, in the sense that everything that can be expressed and deduced in an order-sorted language can also be expressed and deduced in an unsorted language, and vice versa (Walther, 1987). We also introduce in $(ML)^2$ a number of standard axiomatizations of mathematical structures often needed for modelling knowledge in various domains such as: numbers, sets and tuples.

Notice that all of these extensions keep $(ML)^2$ within the framework of first order predicate logic. Many authors in AI have argued in favour of richer logics than just a first-order predicate logic, as witnessed by the plethora of such languages in recent years (epistemic, temporal, uncertain, multi-valued, non-monotonic, etc). Although we do not explore these avenues in this paper, such

³We only discuss the first three layers extensively (domain-, inference- and task-layer). Section 6.1 discusses future work on formalising the strategy layer.

approaches are wholly compatible with the formalism we propose here, and may well prove useful in the KADS framework. Our current formalisation can be seen as entirely parameterised over the choice of the logical language used at the domain layer, very much in the spirit of SOCRATES (Jackson, Reichgelt & van Harmelen, 1989).

2.2 Modularity

If $(ML)^2$ is to be a useful language for modelling large scale applications, we will need a mechanism that allows us to express our theories in a *modular way*⁴. Instead of having to state all our axioms in a single large theory, we split our set of axioms into a number of *sub-theories*, which can then be combined using a small set of meta-theoretic operators. In $(ML)^2$, we currently use set-theoretic union as the only way of combining theories into larger ones, but more sophisticated operators have been proposed in the literature: (Sanella and Burstall, 1983) in the context of an algebraic specification formalism and (Brogi, Mancarella, Pedreschi & Turinin, 1990) for logical theories. Again, the subtheory extension we use here is a conservative one, in the sense that it does not alter the expressive or inferential power of the system over and above a single first-order theory. For an extensive formal investigation into the properties of module systems, see (Bergstra, Heering and Klint, 1990). Below, we will first define the notion of a theory, and its associated syntax. Then we will show how these (sub)-theories can be combined to form a larger theory by means of a meta-theoretic *import* operator.

Formally a *theory* consists of a signature Σ which defines a language plus a set of axioms \mathcal{A} expressed in this language:

$$\mathbf{theory} = \langle \Sigma, \mathcal{A} \rangle \quad (1)$$

The syntax we adopted for a theory is as follows⁵:

```

theory theory-name
  import theory-symbol* ;
  signature
    sorts sort tree ;
    constants (constant-symbol + : sort-symbol) *;
    functions (function-symbol + : sort-symbol +  $\rightarrow$  sort-symbol) *;
    predicates (predicate-symbol + : sort-symbol *) *;
  variables (variable-symbol + : sort-symbol) *;
  axioms axiom *
end-theory

```

This defines a theory with a *theory name*, containing a list of axioms, which are expressed in terms of predicates, functions, constants, and variables, each of which has to obey a type specification. The type of a constant and a variable is specified by a single sort, the type of an n -ary predicate is specified by an n -tuple of sorts, and the type of an n -ary function is specified by an $n + 1$ -tuple of sorts. The *sort tree* is textually expressed as a nested list of sorts, where the structure of the list specifies the subsort relationships. (This notation implies that sorts are only allowed to have one supersort, in other words: the sort hierarchy is a tree, as required in order sorted logic).

⁴A second reason for modularising the domain-layer is related to the way this knowledge is going to be used at the inference layer. This will be further explained in section 3.2.1.

⁵Our syntax for theories is very much inspired by algebraic specification languages (see for instance (Bergstra *et al.*, 1990)). We only give a definition by example here, for a detailed description see (van Harmelen, Balder, Aben & Akkermans, 1991). The keywords in our examples are printed in **bold**. User-definable terms are printed in *italics*. * denotes zero or more repetitions of the term indicated, + denotes one or more repetitions.

The collection of sorts, variables, constants, functions and predicates of a theory is called the *signature* of that theory. Apart from the elements of the signature, the axioms can also use the logical constants of first-order predicate calculus. In order to avoid repetitious typed quantifiers in the expressions, all variables occurring in the axioms are implicitly universally quantified over the specified type. Existential variables or universal variables with a scope smaller than the entire sentence need to be quantified explicitly.

Apart from the signature of a theory, axioms can also use predicates, functions and constants that are defined in the signature of any imported theory⁶. An *import* of theory T_1 into theory T_2 has as a result that the signature and axioms of T_2 become augmented with those of T_1 . Formally:

$$T_1 = \langle \Sigma_1, \mathcal{A}_1 \rangle \text{ import } T_2 = \langle \Sigma_2, \mathcal{A}_2 \rangle \rightarrow T_1 = \langle (\Sigma_1 \cup \Sigma_2), (\mathcal{A}_1 \cup \mathcal{A}_2) \rangle.$$

Standard mechanisms are available from algebraic specification languages to resolve possible name clashes resulting from such *import* operations, by using a *renaming* operation.

2.3 Ontological primitives

Other publications on KADS, e.g. (Wielinga *et al.*, 1992), state that the ontological primitives that are used to describe domain theories are: concepts, property/value pairs and relations, very much in the spirit of concept languages like KL-ONE (Brachman & Schmolze, 1985). It is well known that many elements of such concept languages can be expressed in first order logic. Below, we will show how this is done, and at the same time give a number of small examples of the use of (ML)² at the domain layer of KADS models of expertise.

In our (ML)² models we use order-sorted logic to represent the ontological primitives. Concepts are represented by constants, i.e. nameable and distinguishable entities. Sorts are classes of such entities. The relation between sorts and constants is the equivalent of the IS-A relation in the model of expertise. A tree of IS-A relations maps onto a tree of subsorts. We will illustrate our ideas on the following example:

```

theory carFailure
  signature
    sorts number (vehicle (bus car (station-car limousine))) ;
    constants
      myRolls : limousine ;
      myCar, yourCar : car ;
      bus_412 : bus ;
    functions
      noOfWheels : vehicle  $\rightarrow$  number ;
      noOfReadingLamps : limousine  $\rightarrow$  number ;
    predicates
      sameBrand : car  $\times$  car;
      greater : number  $\times$  number;
  variables a, b, c : number ;
  axioms
    sameBrand(myCar, yourCar);
    greater(a, b)  $\wedge$  greater(b, c)  $\rightarrow$  greater(a, c) ;
    noOfWheels(yourCar) = 4 ;
    greater(noOfWheels(bus_412), noOfWheels(yourCar)) ;
endtheory

```

⁶Note that variables are excluded from this operation. Importing them confused users of (ML)². Generic names like x , y are often used for different variables, and this would be impossible if variables were included in the *import* operation.

Note that the sort/subsort mechanism is a nice way to direct the sharing of properties. The function *noOfWheels* is defined on the most general sort and thus it can be defined for: *myRolls*, *myCar*, *yourCar* and *bus_412*. Thus if a predicate- or function-scheme is defined for a sort it is also defined for all sub-sorts. The predicate *noOfReadingLamps* is conceptually only meaningful for limousines and thus can only be defined on constants of type *limousine*, i.e. for *myRolls* and not for *myCar*, *yourCar* and *bus_412*.

Relations between concepts map straightforwardly onto predicates in the axioms section of our theories. For instance when defining a predicate that expresses that two cars are of the same brand the formal definition is: *sameBrand : car car*. Predicates can also be used for relations between attributes, for instance: *greater(noOfWheels(bus_412), noOfWheels(yourCar))*.

Axioms can contain variables. In (ML)² the syntax of such expressions is: *greater(a, b) ∧ greater(b, c) → greater(a, c)*. The implicit assumption here is that the axiom is quantified over all variables. Existential variables or universal variables with a scope smaller than the entire sentence need to be quantified explicitly.

Attributes of concepts are keyword/value pairs attached to a concept. They map nicely onto functions, whereby the function name represents the keyword. The fact that concepts have a certain attribute is reflected in the declaration of a function scheme. The value of an attribute is reflected in axioms concerning functions: *noOfWheels(yourCar) = 4*.

The above example about ontological primitives consists of only one theory. In order to be able to illustrate the full potential of our approach we will present an example that contains all KADS layers captured in (ML)². The example is small and artificial, but it is large enough to show all the elementary pieces of our formalism at work. The inference process modelled in the example contains two steps: an abductive one where the hypothetical cause for a failure symptom is found, and a deductive one where an observable verification for the possible cause is generated.

2.4 Example domain layer

In the domain layer of our example, we find five theories. The two main theories T_1 and T_2 define the actual domain knowledge and T_Σ defines a signature common to T_1 and T_2 . The fourth theory defines knowledge about the current case. The last one will eventually contain the result of the inference process. First we define the common signature. Notice that this example uses only propositional language at the domain layer. The argument to all functions and predicates, i.e. the particular car with failure, is omitted because it carries no extra information. A complete version of the example can be found in (van Harmelen *et al.*, 1991).

```
theory TΣ
  signature
    sorts reading ;
    constants low, zero : reading ;
    functions
      gasDial : → reading ;
      voltageDial: → reading ;
    predicates batteryLow; engineDoesntRun; noGas;
endtheory
```

T_1 contains axioms that express possible causes for faults in the car-engine domain.

```
theory T1
  import TΣ ;
  axioms
    batteryLow → engineDoesntRun ;
```

```

        noGas → engineDoesntRun ;
endtheory

```

T_2 contains axioms about observable phenomena for certain causes.

```

theory  $T_2$ 
  import  $T_{\Sigma}$  ;
  axioms
    batteryLow → voltageDial = low ;
    noGas → gasDial = zero ;
endtheory

```

Theory `caseData` specifies the data specific for the current case, namely the fact *engineDoesntRun*.

```

theory caseData
  import  $T_{\Sigma}$  ;
  axioms engineDoesntRun ;
endtheory

```

Finally we define a theory that will contain the result of the inference process. Initially this theory is of course empty. Notice that theory `result` does define the necessary signature to express this result.

```

theory result
  import  $T_{\Sigma}$  ;
endtheory

```

This concludes the presentation of the $(ML)^2$ fragment used to represent the domain layer of a KADS model. The same constructions will also be used at the other layers of a KADS model, that will be discussed in the next sections, although these other layers will involve additional constructions that are not used at the domain layer.

3 Inference layer

The purpose of the inference layer of a KADS model of expertise is to state what the potential inferences are that can be made by using the knowledge specified at the domain layer. Note that the inference layer does not state which inferences must actually be made, or in which order. Each of these possible inferences is modelled by a *knowledge source* (a primitive inference action). Such knowledge sources are characterised by a unique name (such as `abstract`, `generalize`, etc.), and are regarded as atomic entities with no further internal structure. A knowledge source operates on data-elements and produces a new data-element. These data-elements are modelled at the inference layer by *meta-classes*. Meta-classes describe the *roles* that elements of the domain knowledge play in the inference process. For instance, certain domain expressions will be used as an `observable`, others as a `hypothesis` or an `assumption`, and each of these roles will correspond to a meta-class that can be used as the input to knowledge sources, or produced as the output of them.

Thus, a crucial aspect of the KADS inference layer is the description of *roles* of domain expressions, and to specify how these expressions are to be used in the inference steps. In other words, the inference layer is a theory *about* the domain layer, namely about the use of the domain layer. Technically speaking, this makes the inference layer a *meta-layer* of the domain layer. Before describing how the inference layer can be represented in $(ML)^2$, we will give a brief overview of some standard constructions in meta-logic that will be used in $(ML)^2$.

3.1 Introduction to meta-logic

3.1.1 Object-meta theories

The language of logic allows us to express properties of and relations between individuals in a domain of discourse. For instance, in the domain of natural numbers, we can state that a number is even, or that a number is the sum of two other numbers. However, the language of first order logic does not allow us to state properties of relations, or properties of properties. For instance, if we want to state what it means for a relation to be commutative, we would need to say:

$$\forall p : \text{commutative}(p) \leftrightarrow \forall x \forall y : p(x, y) \rightarrow p(y, x)$$

which is a second order sentence, and thus outside our first order language.

In order to enable the expression of general properties of logical theories and of expressions in such theories, logicians have formalised the notion of a *meta*-theory. A meta-theory \mathcal{M} is a theory (some of) whose terms refer to formulae of another theory \mathcal{O} , called the *object*-theory. This construction allows the formulation in \mathcal{M} of general properties of (formulae of) \mathcal{O} , while staying within a first order language. The semantics of this construction was already investigated in (Tarski, 1936), who showed that the object-theory \mathcal{O} must be seen as a (partial) model of the meta-theory \mathcal{M} .

Meta-constructions have a longstanding tradition in both logic and AI. See (Feferman, 1936) for some of the early work in logic on reflective systems, (Weyrauch, 1980) and (Smith, 1984) for some early reflective systems in AI, (Giunchiglia & Smail, 1989) for a good introduction to the terminology and the literature, and (Maes & Nardi, 1988) for a collection of some recent work in this area.

Two aspects are crucial to the construction of such pairs of object-meta theories. The first one concerns naming: as mentioned before, terms of \mathcal{M} will need to refer to formulae of \mathcal{O} . Such terms in \mathcal{M} are called *names* of the formulae in \mathcal{O} . The second aspect concerns reflection rules. Inference in one of the theories \mathcal{M} or \mathcal{O} will need to be connected to inference in the other theory, so that deduction in one theory is affected by deduction in the other. This is established through special inference rules called *reflection rules*. Each of these aspects will be described in some more detail below.

3.1.2 Naming

In order to enable \mathcal{M} to express properties of \mathcal{O} , \mathcal{M} will need to have names for formulae in \mathcal{O} , so that the properties can be expressed in terms of these names (and therefore in terms of first order predicates). Tarski (1936) distinguished two types of names: *quotation names*, where a formula ϕ from \mathcal{O} is named by an atomic constant from \mathcal{M} , often written as $[\phi]$, and *structural names*, where a formula ϕ from \mathcal{O} is named by a complex term from \mathcal{M} , with the structure of the term from \mathcal{M} corresponding to the syntactic structure of the formula ϕ from \mathcal{O} . Such naming relations were always (and often tacitly) assumed to be *total* (each formula in \mathcal{O} has a name in \mathcal{M}), *injective* (no two formulae in \mathcal{O} have the same name in \mathcal{M}) and *functional* (each formula in \mathcal{O} has at most one name in \mathcal{M}).

The naming relations as traditionally used by logicians (and in many AI systems) are always purely *syntactical*, in the sense that syntactically similar formulae from \mathcal{O} have syntactically similar names in \mathcal{M} (either because all names are atomic constants, or because their names have a structural correspondence with the syntax of the formulae in \mathcal{O}).

In (Akkermans *et al.*, 1990) and (van Harmelen *et al.*, 1990, chapter 3), we have argued that it is useful to consider naming relations that are not only based on the syntax of the formulae in

\mathcal{O} , but also on the semantics (meaning) and pragmatics (use) of these formulae. Such naming relations, which we called *meaningful naming relations*, must of course be defined separately for each object-theory, because of their dependence of the semantics and pragmatics of each specific theory. Such meaningful naming relations will be used in section 3.2.1 in the form of *lift-definitions* to represent meta-classes, which model the *use* of domain expressions in the inference process.

3.1.3 Reflection rules

Pairs of object-meta theories can be equipped with a special type of inference rule, namely a rule whose premise is in one theory, and whose conclusion is in the other. Such rules are called reflection rules, and establish a connection between deduction in the two theories of a meta-object pair. The examples of reflection rules that are most often encountered in the literature are:

$$\frac{\vdash_{\mathcal{O}} \phi}{\vdash_{\mathcal{M}} \text{prove}(\mathcal{O}, [\phi])} \quad (\text{up}) \qquad \frac{\vdash_{\mathcal{M}} \text{prove}(\mathcal{O}, [\phi])}{\vdash_{\mathcal{O}} \phi} \quad (\text{down})$$

The first of these rules, called upwards reflection, states that provability of ϕ in \mathcal{O} entails the provability of $\text{prove}(\mathcal{O}, [\phi])$ in \mathcal{M} , and allows deduction in \mathcal{M} on the basis of deduction in \mathcal{O} . The second rule, called downwards reflection, states the reverse of rule (*up*), and allows deduction in \mathcal{O} on the basis of deduction in \mathcal{M} . In both these rules, the term $[\phi]$ in \mathcal{M} is the name of the formula ϕ in \mathcal{O} , in the sense described in the previous section.

Although the rules (*up*) and (*down*) are the reflection rules most often encountered in the literature, they are by no means the only examples of reflection rules. Instead of these rules that concern provability in \mathcal{O} , we can also write down rules concerning axiomhood of \mathcal{O} :

$$\frac{\phi \in \mathcal{O}}{\vdash_{\mathcal{M}} \text{ask}_{\varepsilon}(\mathcal{O}, [\phi])} \quad (\text{ask}) \qquad \frac{\vdash_{\mathcal{M}} \text{tell}(\mathcal{O}, [\phi])}{\phi \in \mathcal{O}} \quad (\text{tell}) \qquad (2)$$

where $\phi \in \mathcal{O}$ is used as notation for ϕ being an axiom of \mathcal{O} . The distinction between axiomhood and theoremhood is never made in the logical literature, and indeed there is no need to do so on purely logical grounds, but on both epistemological and computational grounds there are good reasons to distinguish between an axiom of a theory and a derived conclusion of a theory, and thus to consider rules (*ask*) and (*tell*) as well as rules (*up*) and (*down*). (van Harmelen, 1991) discusses a collection of reflection rules that model a large number of other properties of \mathcal{O} that may be of interest in meta-theoretic constructions, but that are not currently part of (ML)².

3.2 Meta-logic in relation to (ML)²

In this section we will explain how the meta-constructions described in the previous section can be used to model the KADS inference layer. As said before, the constituents of an inference layer are meta-classes, knowledge sources, plus the way they are connected. We will treat each of these in turn:

3.2.1 Meta-classes as lift-definitions

The main function of a meta-class is to describe a role of domain expressions to indicate their use in the inference process. It is therefore natural to represent the meta-classes as *naming operations*, where the names of domain expressions will encode the role the expressions will play in the inference process. In order to enable the definition of such naming relations, (ML)² contains *lift-definitions*.

These define a mapping called *lift* from the language of an object-theory to ground terms in a meta-theory:

$$\mathbf{lift}: \mathcal{L}_{\mathcal{O}} \rightarrow \mathcal{L}_{\mathcal{M}}^{ground}.$$

We specify such a *lift-definitions* as follows:

```

lift-definition meta-class-name
  from object-theory-name * ;
  to meta-theory-name *;
  use lift-definition-name *;
  signature
    sorts sort tree;
    constants (constant-symbol + : sort-symbol) * ;
    functions (function-symbol + : sort-symbol + → sort) *;
  lift-variables (var-symbol + : metatype) *;
  mapping (lift(obj-theory, exp) ↦ term) *
end-lift-definition

```

We see that a lift-definition defines a variable-free signature⁷ (the one corresponding to the mentioned meta-language of ground terms), plus a set of rewrite rules that define the behaviour of the *lift* mapping. This signature can then be used by other (meta-level) theories (specified in the *to* list), allowing them to refer to terms from the object-level theory.

The mapping of a lift-definition defines the behaviour of *lift* on the languages of theories mentioned in the *from* list. The rules are of the form

$$\mathbf{lift}(\mathit{theory}, e_1) \mapsto e_2,$$

where *theory* is one of the theories mentioned in the *from* list, e_1 is an expression in the language defined by the signature of *theory* plus the specified lift-variables, and e_2 is a term in the language defined by the lift-definition's signature plus the lift-variables occurring in e_1 . The lift-variables are introduced to allow the expression of mapping rules via schemas. They must be of a *metatype* (one of: variable, constant, function, term, predicate or sentence, organised in the obvious type hierarchy, and the second-order types function symbol, predicate symbol or logical symbol), and range over the corresponding expressions in the language of the object-level theory. Thus, *lift* rules can be expressed either by specifying the behaviour of *lift* on a particular expression of the object-level language (when e_1 is a ground expression of the object-level theory), or by specifying its behaviour on a class of expressions of the object-language (when e_1 contains *lift-variables*) by means of schemas. When syntactic schemas do not discriminate between domain knowledge, and we still want to avoid having to define *lift* rules on individual ground expressions, we can try to modularize the domain knowledge in such a way that expressions that have identical syntactical properties but different roles are divided into different theories. In this way, the modularisation of the domain layer is influenced by the use of this knowledge at the inference layer.

As mentioned before, the main function of a meta-class is to describe the role that domain expressions play in the inference process. As can be seen in, for instance, lift-definition `symptom` defined in the example in section 3.3 these roles are typically indicated by the function symbols that are introduced in the lift rules. For example the function-symbol `symptom` indicates the role that the domain notions from theory `caseData` will play.

Traditionally, meta-classes are seen as a kind of “storage” that holds either input-knowledge or knowledge that is the result of the inference process. As a result of the development of our formal

⁷The *lift-variables* are not part of the signature.

framework we have come to regard them somewhat differently. The semantics of meta-classes in our formal framework is that of an *interface definition* for knowledge sources. This interface takes the form of the signature of a lift-definition. Note that a lift definition contains only a signature and lift rules, and cannot function as a “storage” device. As will be discussed in section 4.2.1 the storage function is associated with the knowledge sources. To summarize we can say that meta-classes define the interface to knowledge sources, not the actual knowledge that goes in (comes out) of it.

3.2.2 Knowledge sources as theories

A knowledge source is an operation that computes output elements on the basis of input elements. This suggests that a knowledge source must be represented as a relation between its input and output meta-classes, and thus as a predicate $KS_k(t_1, \dots, t_n)$, where each t_i corresponds to one of the input meta-classes of the knowledge source, or to the output meta-class. The predicate KS_k , called a *knowledge source predicate*, specifies the relation that holds between inputs and output of a knowledge source. $KS_k(t_1, \dots, t_n)$ is intended to be true *if* the corresponding knowledge source transforms the “input” t_1, \dots, t_{n-1} into the “output” t_n . The definition of a knowledge source then consists of a theory containing axioms which specify what this relation between input and output is. Such axioms will look like:

$$LHS_{KS_k} \rightarrow KS_k(t_1, \dots, t_{n-1}, t_n) \quad (3)$$

The left-hand side LHS_{KS_k} can be an arbitrary formula constructed from predicates of the form $input_{MC_i}(t_i)$, and each t_i will be a term whose outermost function symbols comes from the signature of the lift-definition that represents meta-class MC_i . The arguments of this outermost function symbol can be either variables or function-values occurring in other $input_{MC_i}$ expressions. We will not provide a definition for the $input_{MC_i}$ predicates here, but will postpone this to section 4.2. This will give rise to a differentiation of meta-classes.

A valid example of the definition of a *knowledge source predicate* is for instance:

$$\begin{aligned} &input_{symptom}(symptom(Y)) \wedge \\ &input_{causeModel}(causeFor(X, symptom(Y))) \rightarrow \\ &KS_{abstract}(symptom(Y), causeFor(X, symptom(Y)), cause(X)) \end{aligned}$$

under the assumption that $symptom()$ is a function defined in meta-class **symptom**, $causeFor()$ is defined in input meta-class **causeModel** and $cause()$ is defined in the output meta-class.

The above definition of the general form of knowledge source predicate (3) is in fact too strict. Also axiom sets that are equivalent to the above definition are adequate. For instance axiom sets of the form

$$\begin{aligned} LHS_{KS_{k1}} &\rightarrow A(t_1, \dots, t_i) \\ LHS_{KS_{k2}} &\rightarrow B(t_{i+1}, \dots, t_{n-1}) \\ A(t_1, \dots, t_i) \wedge B(t_{i+1}, \dots, t_{n-1}) &\rightarrow KS_k(t_1, \dots, t_{n-1}, t_n) \end{aligned}$$

which introduce some intermediate predicates (A and B) for epistemological reasons.

It is also allowed to specify more defining clauses for the *knowledge source predicate*. For instance one can define multiple clauses depending on certain preconditions.

3.2.3 Connecting meta-classes and knowledge sources

So far we have introduced the formal constructions that model the principal building blocks of the inference layer: the meta-class, modelled by the lift-definition, and the knowledge source modelled by a theory. We did not yet define how we combine these elements into an inference structure. Connections in a KADS inference structure are merely input/output relations for a knowledge source that are denoted by arrows in the inference structure. Formally we define these connections as follows. Remember that the formal equivalents of meta-classes (lift-definitions) primarily introduce a language. The meaning of a particular meta-class being input (or output) to a knowledge source is that the knowledge source can use the language elements of this meta-class in its axioms (as explained in the previous section). Thus we need a meta-theoretic operation similar to *import*, defined in section 2.2, to introduce the language of meta-classes (lift-definitions) into a knowledge source (theory)⁸. This is done through a *use* operator clause in the meta-level theory. When a lift-definition is used in a meta-level theory, we construct the union of the signature of the theory and the (meta-)signature produced by the lift-definition. Formally:

$$\langle \Sigma_1, \mathcal{A} \rangle \text{ use } (\mathcal{L}_O \xrightarrow{\text{lift}} \mathcal{L}_M^{ground}) = \langle (\Sigma_1 \cup \Sigma_{\mathcal{L}_M^{ground}}), \mathcal{A} \rangle,$$

where $\Sigma_{\mathcal{L}_M^{ground}}$ is the signature underlying the language \mathcal{L}_M^{ground} .

Unfortunately, the *use* operators do not distinguish between input and output connections in the inference structure (as indicated by the direction of the arrows in a normal inference structure). We can no longer see whether a meta-class is input or output to a knowledge source. Remember, however, that this information is still present in the axioms in the knowledge source that define the *knowledge source predicate* (section 3.2.2).

We now have defined formal constructs that can be used to build a formal model of a KADS inference layer. We introduced so-called *lift-definitions*: these are formal constructs that define a inference-level naming for domain-level constructs to model meta-classes. For modelling knowledge sources we introduced *knowledge source predicates*. We continue our example of the car-failure domain by presenting the inference layer of this example.

3.3 Example inference layer

We will illustrate the (ML)² constructions for the inference layer in the car-failure example, by defining an inference layer for the domain theories that were given in section 2.4. Before we give the meta-classes (lift-definitions) and knowledge sources (theories), we show in figure 3 the inference structure and its relation to the domain layer. Strictly speaking the theories presented in the inference layer are not correct (ML)²: we have suppressed the information related to sorts, since it is not relevant to the discussion of the inference-layer of (ML)², and would only complicate the presentation of the example.

The lift-definitions are defined one for each meta-class (as specified in section 3). The first one (**symptom**) gives a role to the knowledge in theory **caseData**.

```
lift-definition symptom
  from caseData ;
  to abstract ;
  signature
    constants [P] ;
```

⁸Sometimes we also need to *use* the language defined in one lift-definition in another. This operation is defined similarly as done here for a theory.

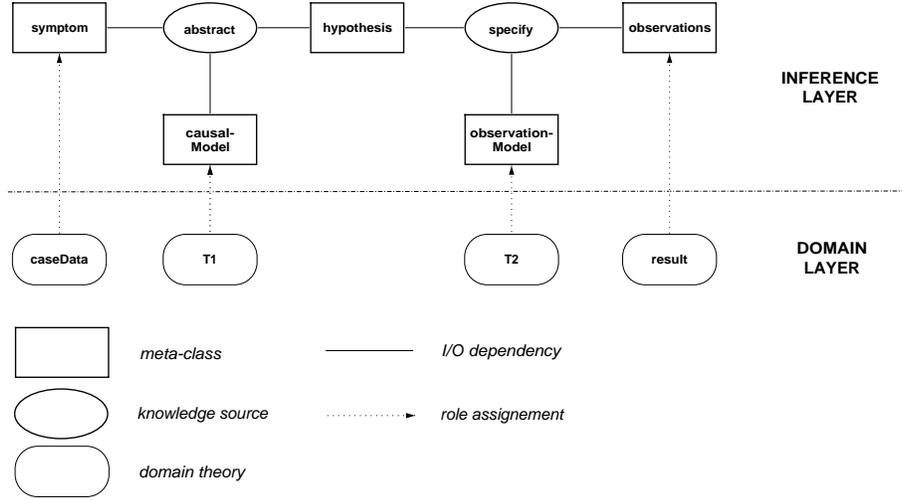


Figure 1: Inference structure for example

```

functions symptom ;
lift-variables P : predicate ;
mapping lift(caseData, P)  $\mapsto$  symptom([P]) ;
end-lift-definition

```

Next, `causeModel` lifts the knowledge about possible causes. Notice that we need to *use* lift-definition `symptom` because we need the function `symptom` in our mapping. Strictly speaking, `causeModel` is not a meta-class since it contains only permanent domain knowledge. (Wielinga *et al.*, 1992) introduce the term *domain views* for these constructions, rather than meta-classes. The same formal machinery as used here could be used to model this point of view.

```

lift-definition causeModel
from T1 ;
to abstract ;
use symptom ;
signature
  constants [P2] ;
  functions causeFor ;
lift-variables P1, P2 : predicate ;
mapping lift(T1, P1  $\rightarrow$  P2)  $\mapsto$  causeFor([P1], symptom([P2])) ;
end-lift-definition

```

The first knowledge source uses the causal relations to establish causes for observed failures. This is an abductive inference step. Remember that the *input...* predicates will be defined at the task-layer. The “output” of this knowledge source is specified in terms of the signature defined in its output meta-class (`hypothesis`).

```

theory abstract
use hypothesis, causeModel, symptom ;
signature
  predicates KSabstract ;
variables X, Y ;
axioms
  inputsymptom(symptom(Y))  $\wedge$ 
  inputcauseModel(causeFor(X, symptom(Y)))  $\rightarrow$ 
  KSabstract(symptom(Y), causeFor(X, symptom(Y)), cause(X))
endtheory

```

We still need to define the meta-class in between the two knowledge sources. This is an internal meta-class. Such meta-classes are also modeled by lift-definitions (to introduce a new signature), but do not contain any mapping rules.

```
lift-definition hypothesis
  to abstract, specify ;
  signature
    functions cause ;
end-lift-definition
```

Lift-definition `observationModel` lifts knowledge about observables for causes. Again, this would correspond to a domain view, according to (Wielinga *et al.* 1992) . As `causeModel`, this lift-definition *uses* the signature of another lift-definition (`hypothesis`).

```
lift-definition observationModel
  from T2 ;
  to specify ;
  use hypothesis ;
  signature
    constants [P1], [P2] ;
    functions observedBy ;
  lift-variables P1, P2 : predicate ;
  mapping lift(T2, P1 → P2) ↦ observedBy(cause([P1]), [P2])
end-lift-definition
```

Now follows the second knowledge source, which finds observables that should verify the potential cause. This is a simple deductive step. Again the “output” is expressed in terms of the output meta-class `observable`.

```
theory specify
  use observable, observationModel, hypothesis ;
  signature
    predicates KSspecify ;
    variables X, Y ;
  axioms
    inputhypothesis(cause(X)) ∧
    inputobservationModel(observedBy(cause(X), Y)) →
    KSspecify(cause(X), observedBy(cause(X), Y), observable(Y))
endtheory
```

The last meta-class will be used to channel the output of the inference process back to the domain layer.

```
lift-definition observable
  from result ;
  to specify ;
  signature
    constants [P] ;
    functions observable ;
  lift-variables P : predicate ;
  mapping lift(result, P) ↦ observable([P])
end-lift-definition
```

4 Task layer

The purpose of the task-layer in a KADS model of expertise is to enforce *control* over the inference steps specified at the inference layer. The inference layer specifies which *possible* steps can be

taken, and what their dependencies are, but it does not specify in which *order* these steps should be executed. This is the concern of the task layer.

The formal framework used so far to describe the domain and inference layer of a formal model uses a language that can be characterized by the following pseudo-equation

$$(\text{ML})^2 = \text{FOPC} + \text{sub-theories} + \text{lift} + \text{reflective predicates.}$$

This language does not contain any vocabulary for capturing the notions related to control that are involved in the *task layer* of a model of expertise. We extend the $(\text{ML})^2$ language used so far with *dynamic logic*, a logic specially designed to describe programs, and use the constructions of dynamic logic to describe a KADS task layer.

In this section we will first give a short overview of Quantified Dynamic Logic (QDL). Then we will show how to use QDL to describe notions in the KADS task layer, i.e. how to use the declarative descriptions of knowledge sources procedurally. Next we introduce some definitions that simplify the specification of tasks. Finally we describe how tasks can be specified.

4.1 Introduction to Quantified Dynamic Logic

Quantified Dynamic Logic (QDL) is an extension of first order logic developed for reasoning about properties of programs. The account of QDL we give here is mainly based on (Harel, 1984). We give the main intuitions behind QDL in the main body of the text, but do not give many of the formal details. These can be found in (Harel, 1984).

4.1.1 Syntax

The main ingredients of QDL are: predicate logic, atomic programs, and syntactic constructions that allow the expression of sequence, condition, iteration, etc. QDL differs from ordinary logic because it introduces programs, Harel (1984) gives the following intuitive notions about programs:

variable a variable is a (named) storage that can hold a value. In contrast to ordinary logic a variable may assume different values during the execution of a program.

state ...at any point in the computation, the program operates on an *execution state* determined by the current value of all its variables. A program thus can be conceived as a transformation between pairs of such states. ...One might say that it *transforms* the *initial state* into the *final state*.

QDL introduces one kind of atomic program, that is the assignment statement: if x is a variable ($x \in V$, V the set of variables) and σ is a term, then $x := \sigma$ is an assignment statement. Other ingredients are make compound programs out of atomic ones (written “;”, “ \cup ” and “ \star ”), and a construction to turn logical formulae into programs (written as ?). Intuitively, the compound programs have the following intended meanings (where ϕ denotes a predicate and α and β denote programs):

$\alpha; \beta$ (pronounced “ α then β ”): do α followed by β ,

$\alpha \cup \beta$ (pronounced “choose α or β ”): do either α or β , nondeterministically,

$\alpha \star$ (pronounced “repeat α ”): repeat α a finite, but nondeterministically determined, number ≥ 0 times,

$\phi?$ (pronounced “test ϕ ”): proceed if ϕ is true, else fail.

With these elementary constructs we can define various programming constructs that are known from programming languages. For instance:

<i>if ϕ then α else β</i>	as	$(\phi?; \alpha) \cup (\neg\phi?; \beta)$
<i>while ϕ do α</i>	as	$(\phi?; \alpha)^* ; \neg\phi?$
<i>repeat α until ϕ</i>	as	$\alpha ; (\neg\phi?; \alpha)^* ; \phi?$
<i>case $\phi : \alpha, \psi : \beta$</i>	as	$(\phi?; \alpha) \cup (\psi?; \beta)$

The final new ingredient of QDL is a modal operator $\langle \alpha \rangle$ for every program α . The compound formula $\langle \alpha \rangle \phi$ has the following intended meaning: *it is possible to execute α reaching a state where ϕ is true*. In other words: ϕ is true in at least one terminal state of α . We abbreviate $\neg \langle \alpha \rangle \neg \phi$ to $[\alpha] \phi$ which is intended to mean: *ϕ is true in all terminal states of α* . In other words: after α , ϕ is necessarily true. Thus, associating a modal operator $\langle \alpha \rangle$ with each program α makes QDL a multi-modal logic, with infinitely many modal operators.

4.1.2 Semantics

The semantics of dynamic logic is a modal one, where “possible world” is characterised by the values of all the variables (also known as a “state”), atomic programs are transitions between states, and atomic formulae are assigned a truth value in each state. Thus, the meaning of an expression like $\langle \alpha \rangle \phi$ is: there is a state s (characterised by the values of all the variables) such that s can be reached by executing α (i.e. the execution of α results in assigning exactly those values to all the variables that characterise state s), and ϕ is true in state s (i.e. ϕ is a statement about the values of the variables that is true whenever all the variables have values as defined in state s).

4.2 QDL in relation to (ML)²

We now explain how to apply the machinery of QDL to the task-layer of our formal models. As stated above, the purpose of the task layer is to enforce control over the inferences that are specified in the inference layer. The inference layer specifies which inferences are available as the knowledge sources, and what their dependencies are (“input/output” relations) in terms of the meta-classes. The purpose of the task layer is then to specify in which order these inferences are to be “executed” to achieve a certain goal. We will therefore use the declaratively specified knowledge sources as programs that can be executed, and use the meta-classes as inputs and output of such programs.

4.2.1 Knowledge sources as programs

As described in section 3.2.2, each knowledge source of the inference layer is specified by an “knowledge source predicate”, $KS_i(I_1, \dots, I_n, O)$ ⁹, where the I_1, \dots, I_n are the input values, and the O is the output value. There are as many I_j ’s as there are input meta-classes to the knowledge source. There is only one O since KADS specifies that a knowledge source has exactly one output meta-class. The knowledge source predicate KS_i specifies the relation that holds between inputs and output of an knowledge source, and which is defined by axioms of the form given in (3). QDL gives us precisely the machinery needed to turn such a declarative specification of an I/O relation into an executable program, namely via the test-operator $?$. According to QDL, the expression $KS_i(\vec{I}, O)?$ corresponds exactly to the program specified by the input/output relation $KS_i(\vec{I}, O)$.

⁹From now on, we will use the notation \vec{I} as an abbreviation for I_1, \dots, I_n .

To be able to assess the status of a knowledge source we need to associate one or more variables to it. We choose to associate one such variable, which we call V_{KS_i} , to each knowledge source. In this variable we can store the current status, i.e. a tuple consisting of the inputs and output of the most recently computed inference: $\langle I_1 \dots I_n O \rangle$. We will call this variable the *knowledge source variable*. It will turn out, however, (in section 4.2.2) that for the type of information we want to describe at the task-layer, it is not only necessary to represent the “current” status of the *KS*, (i.e. the most recently computed I/O pair), but instead we want *all* previously computed I/O pairs. Consider for instance the specification of a generate-and-test task-layer that iterates between two knowledge sources, the first one generating possible solutions, and the second one testing to see if they are actual solutions. In such a case, we do not only want to call the *generate* knowledge source to produce a solution, then store this solution in the output meta-class and call the **test** knowledge source, but we want to ensure that each time we call the **generate** knowledge source, we get a *new* solution, that has not been generated before. In other words, we need to represent the *history* of the inference machinery, which consists of all previously computed I/O pairs.

QDL does not by itself give us access to previously computed results. All we can access in QDL is the *current* state of the computation (i.e. the most recently computed I/O pair), but not *previous* states of the computation. QDL gives us only one type of atomic program, namely the assignment statement ($:=$), and it assumes the existence of an equality predicate over terms ($=$). In order to incorporate history in our knowledge source variables, we define a non-atomic program $:\dot{=}$ (pronounced “prepend”) and a predicate $\dot{=}$ (pronounced “contains”) in terms of the atomic operations defined in QDL:

$$(x :\dot{=} y) \equiv (x := \langle y|x \rangle) \quad (4)$$

$$(x \dot{=} y) \leftrightarrow (\exists u : x = \langle y|u \rangle \vee (\exists u, v : x = \langle u|v \rangle \wedge v \dot{=} y)) \quad (5)$$

that correspond to assignment to (the front of) a tuple, and testing for membership of a tuple¹⁰.

By using the new “prepend” operator, the value of a knowledge source variable is going to be a sequence of pairs of the form

$$V_{KS_i} = \langle \langle \vec{I}_1, O_1 \rangle, \dots, \langle \vec{I}_n, O_n \rangle \rangle$$

Summarizing we can say that the predicate $KS_i(\vec{I}, O)$ specifies which *possible* input/output pairs satisfy the knowledge source specification, whereas the value of the variable V_{KS_i} is going to store the *actual* pairs that have been computed by successive “calls” of the program $KS_i(\vec{I}, O)$?

We want to stress that the formal framework does not, in itself pose any restriction on the “direction” in which a knowledge source is used. This depends on the way we activate the knowledge source. For instance if we activate a knowledge source with the construction:

$$KS_{example}(I_1, I_2, [hypothesis(fever)])$$

(with I_1 and I_2 denoting variables) we actually ask the question: which input (i.e. I_1, I_2) would give the output $[hypothesis(fever)]$ and we would thus be using the knowledge source in the reverse direction.

4.2.2 Task layer primitives

With these definitions, we can now construct some interesting building blocks for the task layer. The first three (*has-solution*, *old-solution* and *more-solutions*) will be predicates (which can of course be turned into programs using the ?-test construction from QDL). The fourth building block

¹⁰ $\langle u|v \rangle$ is the usual notation for $\langle u, v_1, \dots, v_n \rangle$ where $v = \langle v_1, \dots, v_n \rangle$.

(*give-solution*) will be a program. Consequently, the first three tools do not alter the state of the inference machine (predicates and tests do not cause a state-transition), whereas the fourth one does.

The following building blocks are intended to be the *only* ways in which the task-layer can interact with the inference layer. For each knowledge source KS_i , we will define three new predicates and a program to allow interaction with that knowledge source, as follows:

has-solution Sometimes we want to ask a knowledge source whether a solution exists without actually changing the state of the inference machine. This we would do with:

$$has-solution-KS_i(\vec{I}, O) \leftrightarrow KS_i(\vec{I}, O)$$

Notice that *has-solution- KS_i* gives a solution. This solution has no connection with previously obtained solutions, in the sense that it can be the same or a different solution. All that *has-solution- KS_i* does is check if a certain I/O pair satisfies the specification of the knowledge source KS_i . With different instantiations of this predicate, we can of course find out if an output exists for some given inputs, or if inputs exist that would compute a given output, or test a given input/output pair. This predicate can also be used to get all possible solutions or the number of possible solutions.

old-solution Sometimes we want to find out whether some relation has already been computed in the inference process. If so we will find it in the corresponding knowledge source variable:

$$old-solution-KS_i(\vec{I}, O) \leftrightarrow V_{KS_i} \doteq \langle \vec{I}, O \rangle$$

Again, this can be used for a number of purposes: to find out *all* previously computed relations, or the number of them. We can also compute previously computed outputs (instead of I/O relations), or find out what the first, last, or n-th computed relation was. We can also test if a given output has already been computed, or find out what inputs were used to compute it, etc.

more-solutions Using the two predicates defined above, we can find out if there still exist previously uncomputed solutions:

$$\begin{aligned} more-solutions-KS_i(\vec{I}, O) \leftrightarrow \\ has-solution-KS_i(\vec{I}, O) \wedge \neg old-solution-KS_i(\vec{I}, O) \end{aligned}$$

which reads: is there a solution not already computed¹¹. Again, multiple uses allow one to find out the number of uncomputed solutions, all uncomputed solutions, uncomputed solutions for given inputs, etc.

give-solution As mentioned earlier, the above constructions only look for solutions, but they are not recorded as “having been computed”, and they are not represented in the state of the inference machinery (as modelled by the knowledge source variables). This actual act of computation is committed by the last of our constructions:

$$give-solution-KS_i(\vec{I}, O) \equiv more-solutions-KS_i(\vec{I}, O)?; V_{KS_i} \doteq \langle \vec{I}, O \rangle \quad (6)$$

This program first checks if there exists a previously uncomputed solution, and if so, stores it in the corresponding knowledge source variable using the newly defined “prepend” atomic program.

¹¹This definition clearly demonstrates that specifications in (ML)² are *not* concerned with computational efficiency. This definition only specifies what should be done but not how.

topology	type	axiom
input, not output	initial	7
input and output	internal	8
output, not input	terminal	9

Figure 2: Types of meta-classes derived from their input/output characteristics

4.2.3 Definition of the *input* predicates

As stated in section 3.2.2, expression (3), each knowledge source theory contains a number of $input_{MC_i}$ predicates, one predicate for each of its input meta-classes. These predicates were left undefined in section 3.2.2 as they depend on the task structure that is defined upon the inference layer. The purpose of this section is to provide a rationale for defining these $input_{MC_i}$ predicates. In order to do this, we must look somewhat closer at the function of meta-classes in models of expertise. This will lead us to distinguish three different types of meta-classes.

In section 3.2.1 we introduced lift-definitions as the formal constructions to model meta-classes. These lift-definitions assign a role to domain-layer terms. They do so purely on the basis of properties of the domain-layer terms itself, e.g. the explicit names of things or certain syntactical characteristics. We will call this type of meta-classes *initial*, as they deliver the input for the inference process. There are meta-classes, however, that assign roles *not* directly on the basis of domain characteristics but as the result of an inference action. This type of meta-classes defines the output of a knowledge source. Here we can distinguish two cases. Either this output is again input for another knowledge source, and it will not be visible at the domain layer. We will call these meta-classes *internal*. Alternatively, the output is *not* input for another knowledge source. In other words it represents the result of the inference process. The result must be transferred to the domain layer. We will label such a meta-class *terminal*.

Consequently internal meta-classes do not have mapping-rules, nor do they need a domain theory to lift from, whereas initial and terminal meta-classes do.¹² The way in which the type of a meta-class is related to its input/output relation is summarised in table 4.

The assignment of these types to meta-classes can only be made at the task layer, when an ordering is enforced on the execution of knowledge sources. For instance, a meta-class can be either initial or terminal, depending on the direction of the inference process (forward or backwards). This is the reason that the definitions of the $input_{MC_i}$ predicates, which depend on this classification, is postponed to the task-layer, even though these predicates themselves already occur on the inference-layer.

Depending on the type of the meta-class, the definition for the $input_{MC_i}$ predicates that were used in (3), is as follows. If meta-class MC_i is of the *initial* type, the predicate $input_{MC_i}$ is defined as follows:

$$\forall x : input_{MC_i}(x) \leftrightarrow ask_{\in}(O, x) \quad (7)$$

where O is the object-theory that is mentioned in the *from*-clause of *lift-definition* MC_i . To enable ask_{\in} to occur in this definition, we include its defining inference rule, rule (*ask*) from section 3.1.3, as an additional inference rule in $(ML)^2$.

If MC_i is an *internal* meta-class we get:

$$\forall x : input_{MC_i}(x) \leftrightarrow \exists \vec{I} : V_{KS_j} \doteq \langle \vec{I}, X \rangle \quad (8)$$

where KS_j is the knowledge source that has MC_i as its output meta-class.

¹²As a result, we could have modelled internal meta-classes as theories instead of as lift-definitions. For reasons of uniformity we preferred to use lift-definitions.

The role of terminal meta-classes is to map results of the inference process back to the domain layer. However, with the formal machinery described so far, inference activity will only result in changes to the state of the inference machinery (as modelled by the set of V_{KS_i} variables), and will not result in any changes to the contents of the theories on the domain layer. We need an additional construction to achieve this effect. For every such terminal meta-class MC_i , we add an axiom of the following form to the task-layer:

$$\langle V_{KS_j} := \langle \vec{I}, O \rangle \rangle tell(T, O)^{13} \quad (9)$$

where KS_j is the knowledge source that has MC_i as its output meta-class, and T is the domain layer theory that occurs in the *from* clause of lift-definition MC_i . This axiom-schema states that there is at least one terminal state of the program $\langle V_{KS_j} := \langle \vec{I}, O \rangle \rangle$ in which $tell(T, O)$ is true, but since the program is terminating and deterministic, it has at exactly one terminal state. Thus, the axiom states that after the program's execution, $tell(T, O)$ will hold.

In order for this definition to have the desired effect on domain layer theories, we need to include the defining inference rule for the *tell* predicate (rule (*tell*) from section 3.1.3) as an additional inference rule in $(ML)^2$. This inference rule allows us to derive in the object-theory (at the domain-layer) that things are true once they have been derived by computation in the task-layer¹⁴.

4.2.4 What is a task?

Using all this language, we can now define what the formalisation of a KADS task is. First of all we have to specify the definitions of the $input_{MC_i}$ predicates for all input meta-classes, plus for all terminal meta-classes, an axiom of the type (9). After this we can write down a KADS task in $(ML)^2$ as a *task-expression*. Such a task-expression is a program-expression in QDL, built out of: the constructions defined in section 4.2.2, the standard composition operators of QDL, QDL's atomic assignment operator $:=$, and possibly some standard operations like set-operations, arithmetic, etc. We will give some examples of task-expressions in section 4.3.

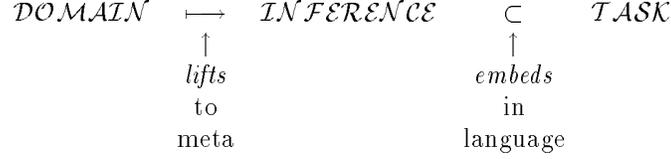
4.2.5 Relation between task layer and inference layer

It is important to notice what the implication of all this formal machinery is for the relation between task- and inference-layer. Remember that the relation between domain- and inference-layer was an object-meta relation, where domain-layer sentences became inference-layer terms that encoded the roles these sentences should play in the inference process, and that were used to specify the useful inferences that could be made using these sentences. The relation between inference- and task-layer is of an entirely different nature: the inference layer is now a sub-theory of the task-layer, expressed in a sub-language (predicate logic) of the language used at the task-layer (dynamic logic). The connection between task-level constructions and inference-level constructions is mainly through the ?-test operation of the task-layer, which turns inference-level predicates into task-level programs. Thus, whereas the domain-inference relation is an object-meta relation, the inference-task layer is an *embedding* relation, in the sense that the predicate-logic expressions of the inference layer are embedded in the language of the task-layer. This in agreement with the intuitive notion of the task-layer *adding* something, i.e. control, to the inference layer.

¹³Notice that the sharp brackets \langle and \rangle have two meanings here: the outer brackets denote the modal diamond operator, the inner brackets denote a tuple.

¹⁴However, see the remarks in in the next section on a variation of this rule that we may require

The situation can be summarised as follows:



4.3 Example task layer

We will now illustrate how to build a task layer on top of the inference structure of the car failure domain given in section 3.3. By virtue of the definitions in section 4.2, we already have the following constructions:

- the variables $V_{abstract}$ and $V_{specify}$, that will become instantiated with values of the form

$$\langle \dots \langle I_1^i \dots I_n^i, O^i \rangle \dots \rangle$$

but initially are both empty sequences;

- definitions for the predicates $has\text{-}solution\text{-}KS_i(I_1, I_2, O)$, $old\text{-}solution\text{-}KS_i(I_1, I_2, O)$, $more\text{-}solutions\text{-}KS_i(I_1, I_2, O)$ and for the programs $give\text{-}solution\text{-}KS_i(I_1, I_2, O)$, as given in section 4.2.2 for $i = \mathbf{abstract}$ and $\mathbf{specify}$.

Given this vocabulary, we can write down a task layer. We will give several examples of a task layer. First we describe a task layer that executes one knowledge source and then the other (example 1). After this we will sketch how tasks can be specified that iterate this process (example 2) or that use only part of the inference structure (example 3). In each of these examples we will comment on the dynamic behavior of the system.

In order to define a task layer we need to construct a single theory with the definitions explained in section 4.2.4. Notice that this is the place where we enforce a direction over the inference structure. This amounts to deciding which meta-classes are *initial* and *terminal*, and then defining their *input* predicates accordingly.

theory *taskDefinitions*

import abstract, specify ;
axioms

$$\text{input}_{\text{symptom}}(X) \leftrightarrow ask_{\in}(caseData, X) \quad (10)$$

$$\text{input}_{\text{causeModel}}(X) \leftrightarrow ask_{\in}(T_1, X) \quad (11)$$

$$\text{input}_{\text{hypothesis}}(X) \leftrightarrow V_{abstract} \doteq \exists A, B \langle A, B, X \rangle \quad (12)$$

$$\text{input}_{\text{observationModel}}(X) \leftrightarrow ask_{\in}(T_2, X) \quad (13)$$

$$\langle V_{specify} \doteq \langle X, Y, Z \rangle \rangle tell(result, Z) \quad (14)$$

endtheory

The predicates in definitions (10), (11) and (13) are defined as ask_{\in} 's because these meta-classes are initial meta-class, whose contents are taken from the domain. The predicate $\text{input}_{\text{hypothesis}}$ is in terms of the knowledge source variable because in this task **hypothesis** is used as an internal meta-class (see section 4.2.3).

Axiom (14) (which is of the form specified in 9) is needed to enable the terminal meta-class **observable** to communicate its contents to the domain layer.

The “real-life” task that we want to model is: given a particular case description, find the cause of a failure and then find an observable to test the cause. In terms of the model of expertise this boils down to executing knowledge source **abstract** followed by executing KS **specify**. We will now give a formal description of this task.

Example 1 (KS execution in sequence) In this example, we will define a task expression for the execution of knowledge source **abstract** followed by the execution of knowledge source **specify**. The following task-expression describes the activation of **abstract**:

$$give-solution-abstract(symptom(Y), causeFor(X, symptom(Y)), cause(X)) \quad (15)$$

This task will execute the knowledge source **abstract** with as its inputs $symptom(Y)$ and $causeFor(X, symptom(Y))$, provide an appropriate binding for X and Y , and will effect the correct instantiation of the $V_{KS_{abstract}}$ variable.

Likewise we can define the execution of **specify** as:

$$give-solution-specify(cause(V), observedBy(cause(V), U), observable(U)) \quad (16)$$

Now we can define the task expression

$$(15); (16) \quad (17)$$

where (15) and (16) refer to the subtask-expressions with these labels mentioned above.

First we will comment on the execution of (15):

1. $give-solution-abstracts(\dots)$ is defined in (6) as

$$(KS_{abstract}(\dots) \wedge \neg V_{abstract} \doteq \langle \dots \rangle?; V_{abstract} \doteq \langle \dots \rangle) \quad (18)$$

which is a sequence of two programs.

2. The first program is a test of two conjuncts. The first conjunct tries to prove $KS_{abstract}(\dots)$ which, by application of the single axiom in theory **abstract** and the task-layer axioms (10) and (11) amounts to proving two ask_{ϵ} expressions:

$$ask_{\epsilon}(caseData, symptom(Y))$$

and

$$ask_{\epsilon}(T_1, causeFor(X, symptom(Y))).$$

3. $ask_{\epsilon}(caseDate, symptom(Y))$ uses the lift-definition **symptom** in the reverse (downwards) to translate $symptom(Y)$ to $[Y]$ ¹⁵. $[Y]$ will match with the only axiom in domain theory **caseData**, i.e. $engineDoesntRun$. Thus Y will become bound to $[engineDoesntRun]$.
4. Because of the binding of Y , the next ask_{ϵ} becomes:

$$ask_{\epsilon}(T_1, causeFor(X, symptom([engineDoesntRun])))$$

It uses the lift-definitions **causeModel** and **symptom** in reverse direction to translate $causeFor(X, symptom([engineDoesntRun]))$ into $[X] \rightarrow engineDoesntRun$, which is indeed an axiom of T_1 if we take for instance $[X] = noGas$, which results in X becoming bound to $[noGas]$. This completes the proof of the first conjunct of the first of the two programs from (18).

¹⁵ $[Y]$ is the “unquoting” of Y : the expression of which Y is the name.

5. The second conjunct of the first of the two programs from (18) is $\neg V_{abstract} \doteq \langle \dots \rangle$, which checks that the argument-triple of the original task-expression (15) does not occur in the value of $V_{abstract}$. Since $V_{abstract}$ is empty (by initialization), this second conjunct succeeds as well. This successfully completes the conjunction that forms the first program from (18).
6. The second program from (18) is $V_{abstract} \doteq \langle \dots \rangle$, which is a simple assignment to the knowledge source variable, to store

$$\begin{aligned} & \langle \langle \text{symptom}([engineDoesntRun]), \\ & \quad \text{causeFor}(\text{cause}([noGas]), \text{symptom}([engineDoesntRun])), \\ & \quad \text{cause}([noGas]) \rangle \rangle \end{aligned} \tag{19}$$

as the new value for $V_{abstract}$.

This terminates the execution of the subtask corresponding to (15). It has resulted in binding the variables X and Y , namely to $[noGas]$ and $[engineDoesntRun]$ respectively, and it has changed the value of the knowledge source variable $V_{abstract}$ as mentioned in (6) above. The execution of (16) is analogous to the execution of (15) as described above. There are, however, some differences. We discuss only those items whose execution is different from above:

2. This step is different because one of the inputs of this knowledge source is an internal meta-class, so the definition of one of the input... predicates is different (compare (10) and (12)). In this case, the first conjunct tries to prove $KS_{specify}(\dots)$ which, by application of the single axiom in theory *specify* and the task-layer axioms (12) and (11) amounts to proving the statements:

$$\text{ask}_{\epsilon}(T_1, \text{observedBy}(\text{cause}(V), U)) \tag{20}$$

and

$$\exists A, B \langle A, B, \text{cause}(V) \rangle \doteq V_{abstract} \tag{21}$$

Note that we need an ask_{ϵ} for the initial meta-class **causeModel**, but that $\text{input}_{hypothesis}$ is defined as an inquiry in a knowledge source variable because **hypothesis** is an internal meta-class.

3. Because of the execution of knowledge source **abstract** the variable $V_{abstract}$ now has the value (19) thus the conjunction of (20) and (21) will succeed with V being bound to $[noGas]$ and U to $[gasDial = zero]$.
6. The second program of (16) is $V_{specify} \doteq \langle \dots \rangle$, which is an assignment to the knowledge source variable, to store

$$\begin{aligned} & \langle \langle \text{cause}([noGas]), \\ & \quad \text{observedBy}(\text{cause}([noGas]), [gasDial = zero]), \\ & \quad \text{observable}([gasDial = zero]) \rangle \rangle \end{aligned}$$

Axiom (14), which is in theory **taskDefinitions** because meta-class **observable** is a terminal meta-class, can be instantiated as:

$$\begin{aligned} \langle V_{specify} \quad \doteq \quad & \langle X, Y, \text{observable}([gasDial = zero]) \rangle \\ & \text{tell}(\text{result}, \text{observable}([gasDial = zero])) \end{aligned} \tag{22}$$

This results in $\text{tell}(\text{result}, \text{observable}([gasDial = zero]))$ being true at the inference level and thus, due to the definition of tell (rule (*tell*) in section 3.1.3), in $gasDial = zero$ being true in domain theory **result**.

In the specification of the system as given above, i.e. before any knowledge source has been executed, we have of course that $gasDial = zero \notin result$. We would expect to be able to prove that after the execution of task expression (15) ; (16) we would have either $gasDial = zero \in result$ or $voltageDial = low \in result$, depending on which possible instantiation for $[X]$ we would have chosen in step 4 above. The problem is however: how should we reason about “the theory **result** after the execution of the task expression”?

To enable this proof to go through, we need to introduce a new meta-theoretic notation, namely: $T^{[\alpha]}$ which stands for “the guaranteed state of theory T after execution of α ”¹⁶. We could use this notation in a modified version of the inference rule (*tell*) from section 3.1.3 for the *tell* predicate as follows:

$$\frac{\mathcal{M} \vdash [\alpha]tell(T, [\phi])}{\phi \in T^{[\alpha]}} \quad (23)$$

stating that if $tell(T, [\phi])$ holds after all possible ways of executing program α , then ϕ is an axiom theory T after execution of α . This rule would then allow us to prove

$$\{gasDial = zero, voltageDial = low\} \cap result^{[(15);(16)]} \neq \emptyset, \quad (24)$$

in other words: at least one of the two object formulae is guaranteed to be an axiom of **result** after execution of (15) ; (16). This proof makes essential use of axiom (14), whose role is precisely to provide a connection between the contents of $V_{specify}$ and the domain theory **result**.

This example shows that execution of a task-expression gives rise to exactly the behavior we expect: knowledge sources are being “activated”, domain-layer knowledge is used for inference, meta-classes are being instantiated, and knowledge is added to the domain layer.

The next example illustrates the use of QDL to specify more complex tasks.

Example 2 (Joint iteration) The task-expression:

$$\begin{array}{ll} \textit{while} & \textit{more-solutions-abstract}(-, -) \\ \textit{do} & (15);(16) \\ \textit{od} & \end{array} \quad (25)$$

will iterate the sequence **abstract** followed by **specify** until **abstract** has produced all its solutions.

It is now possible to prove that

$$\{gasDial = zero, voltageDial = low\} \subseteq result^{[(25)]}, \quad (26)$$

which states that *both* expressions will be axioms of **result** after execution of (25). This is a stronger statement than (24), which only states at least one of them will be an axiom of **result**.

This finishes the first set of example tasks which shows task expressions describing sequences of two knowledge sources. A final example will illustrate how we can use only part of an inference structure in a task. This will require additions to the inference layer, and modifications to the task-theory given above, because the status of a meta-class will change from internal to terminal.

Example 3 (Using an internal MC as terminal) In this example we will execute only the first knowledge source of the inference structure, but, in contrast to example (1), we want the results of this inference to become visible at the domain layer. This will involve changing meta-class **hypothesis** from an internal to a terminal meta-class. In order to achieve this, we need

¹⁶More formally, $T^{[\alpha]}$ is defined by:

$$T^{[\alpha]} \vdash \phi \textit{ iff } T \vdash [\alpha]\phi$$

a mapping of the results of the inference process into domain layer terms, plus an extra theory `intermediateSolution` at the domain layer to store these results.

The required mapping is simply the following mapping rule which must be added to the lift-definition `hypothesis` in the inference layer:

$$\text{lift}(\text{intermediateSolution}, P) \mapsto \text{cause}([P]).$$

Because meta-class `hypothesis` is used as a *terminal* meta-class, we need to add an axiom of the form (9) to the task layer theory:

$$\langle V_{\text{abstract}} := \langle X, Y, Z \rangle \rangle \text{tell}(\text{intermediateSolution}, Z) \quad (27)$$

The result is that any assignments that are made to V_{abstract} as a result of deduction in knowledge source `abstract` will be reflected at the domain layer.

5 Experience

5.1 Application

The ultimate test for each formalism lies in its application. We have applied (ML)² successfully in a dozen formalizations. These all consisted of formalizing previously built models of expertise. The largest of these was the formalization of the model of expertise for making predictions about physical systems by means of qualitative reasoning (Bredeweg, Reinders & Wielinga, 1990). This was a large scale formalization, and resulted in a formal model consisting of 45 theories (24 at the domain layer and 21 at the inference layer) plus 17 lift-operators between domain and inference layer.

Other formalizations have been constructed and published for the cover-and-differentiate problem solving method (Schreiber, Wielinga & Akkermans, 1990), and for heuristic classification (Akkermans, van Harmelen, Schreiber & Wielinga, 1990). Furthermore, formalizations of partial models, i.e. of parts of the inference structure, have been constructed for abduction, and for various forms of abstraction and classification. From these actual formalizations we have learned that, as claimed in the introduction of this paper, formal models reduce the ambiguity of informal models of expertise, provide a precise means of communication about the model of expertise, point out incompleteness and/or inconsistency of the model of expertise.

5.2 Automated support: TheME

Constructing a formal model consists of two subtasks: one is the actual design of the model, the other is more administrative in nature and consists of taking care of all the formal aspects like syntax, declarations etc. This second task is very time consuming and takes a great deal of effort that should have been spent on the actual design task.

The following are common experiences of people building formal specifications:

- building even medium-sized formal models is a task too complex for one person. Often more than one person is needed for the effort, to distribute the tasks of doing the actual design and keeping the model mathematically well-formed.

- several levels of abstraction are used in a single model, often simultaneously. For instance, theories can be either viewed as atomic entities (e.g. elements in an inference structure), or as entities with internal structure (signature, axioms).
- elements of a model are sometimes viewed as atomic elements, while at other times their internal structure is inspected;
- different cross-sections of the model are needed: e.g. a view in terms of KADS concepts, a view in terms of the elements of $(ML)^2$, a view in terms of elements that share a common property etc.

From these observations we decided that it would be worthwhile to have an environment that could support the more formal details of the design. It should enable the user to build a formal model while at all times keeping the model under construction mathematically well-formed. At the same time the environment should ensure that the model represents a KADS model. This must be enforced because $(ML)^2$ in principal allows the expression of constructs that have no semantics within the KADS methodology. We have constructed an environment called `TheME` that monitors all modifications of the model. The effect of each modification is computed and, if the resulting model is still well-formed, the modification is applied. If the modification could result in an ill-formed model, it will not be applied and the user is informed. Besides this insurance of well-formedness, the environment offers support by providing:

- a visual metaphor of the model with which the user can interact: the connectivity of the theories and lift-definitions at the various layers is shown on the screen.
- different points of view: it is possible to view the model in terms of KADS concepts (meta-classes, knowledge sources, layers, etc.) or ML^2 concepts (theories, lift-definitions, imports, etc.).
- different levels of aggregation: entities like theories and lift-definitions can be either viewed as atomic entities, or can be “opened up”, and their contents inspected.
- easy ways of navigating through the model and of finding specific elements: a click and point interface, and on-line indexes of definitions, declarations, etc.
- several ways of outputting the model, either in a form suitable as input for the environment itself, for other tools, or in a form suitable for document preparation.

A more detailed description of the `TheME` environment can be found in (Balder, 1991).

5.3 Mechanization: $Si(ML)^2$

Even though a formal model may be guaranteed to be mathematically well-formed with the aid of an environment like `TheME`, this is of course no guarantee that the model does indeed capture the knowledge and problem solving behavior that the designers set out to capture. Establishing whether or not this is the case is often called *validation* of a model: comparing a specification (formal or otherwise) with the initial requirements of the system to be built. The great advantage of a formal model is that such a validation effort becomes a real possibility, at least in principle. However, in practice the complexity of the formal specification often prohibits many predictions about its actual problem solving behaviour. One solution to this problem, adopted by the software engineering community, is to try and build an interpreter for formal specifications. Such an interpreter can then be used to mechanize the formal model, and to use this mechanization to validate its problem solving behaviour.

In (ten Teye, van Harmelen & Reinders, 1991) we have described an implementation for a subset of $(ML)^2$. This simulator of $(ML)^2$, called $si(ML)^2$, mechanizes a subset of $(ML)^2$. The main restriction on the language implemented by $si(ML)^2$ is the use of Horn Clause logic as the logical language for axioms in theories, instead of full first order logic. As is well known, Horn Clauses are the largest segment of full first order logic that have an unproblematic computational interpretation. Other aspects of $(ML)^2$, notably the multi-layers connected by lift-definitions, are fully implemented in $si(ML)^2$. The $si(ML)^2$ interpreter has been used to mechanize the formal specification of the cover-and-differentiate model of expertise for diagnosis (Eshelman, 1989).

6 Discussion

In this section we will discuss some of the future work that we intend to do on the formal KADS models of expertise.

6.1 Strategy layer

The strategy layer in KADS is concerned, among other things, with constructing tasks that should be executed at the task layer to achieve a certain goal depending on the conditions in which the system finds itself. We expect that this process can also be modeled in $(ML)^2$, namely by a process of theorem proving in QDL.

Expressions of the form

$$\psi \rightarrow \langle \alpha \rangle \phi$$

mean: if ψ is true then ϕ will be true in any state which is a final state of α . Or: under certain initial conditions ψ , if this program terminates, then afterwards ϕ will hold, in other words: given certain preconditions, α may be a way of achieving ϕ .

Expressions of this form can be used to construct complex programs that achieve certain goals starting from certain initial conditions. For example, given the following knowledge at the strategic layer about properties of tasks α_1 , α_2 and α_3 :

$$\begin{aligned} \phi_1 &\rightarrow \langle \alpha_1 \rangle \phi_2 \\ \phi_3 &\rightarrow \langle \alpha_2 \rangle \phi_4 \\ \phi_2 \vee \phi_4 &\rightarrow \langle \alpha_3 \rangle \phi_5 \end{aligned}$$

we can derive that the program:

$$(\phi_1?; (\alpha_1; \alpha_3)) \cup (\phi_3?; (\alpha_2; \alpha_3))$$

is a way of achieving goal ϕ_5 . Thus if we can acquire knowledge about the behavior of programs we can, in principal, dynamically determine what goals are relevant for solve a particular problem. Whether this is a feasible knowledge acquisition task remains to be investigated. Notice that the strategic layer reasons *about* programs, which would make it a meta-layer with respect to the task-layer.

6.2 Task decomposition

In KADS models, a task-layer contains primitive and non-primitive tasks. The non-primitive tasks are decomposed into subtasks, until the level of primitive tasks, i.e. activations of knowledge

sources, has been reached. All these notions can be reconstructed in the formal framework presented above: primitive tasks correspond to activating a knowledge source, which is modeled by the *give-solution-KS* program, while task decomposition can be modeled by constructing compound programs out of constituent programs by means of the program constructors from QDL ($;$, \cup , and \star). For instance, the QDL definition $t_1 \equiv (t_2\star);(t_3 \cup t_4)$ decomposes (the program that corresponds with) task t_1 into (the programs that correspond with) the subtasks t_2 , t_3 and t_4 . It is also possible to formalize alternative task decompositions (i.e. decompositions of the same task into subtasks in different ways) by providing multiple definitions for the same program.

6.3 Customizing (ML)²

In this paper we have indicated which constructions in (ML)² correspond to notions that have a meaning in the KADS methodology. At this moment the builder of the formal model is responsible for staying within these bounds. If an environment like `TheME` is used ((Balder, 1991), and section 5.2 above), certain non-KADS constructions are no longer possible to construct, but others can still be made. A possible solution would be to define a macro-like structure on top of (ML)² that would allow only KADS-constructs. Another useful effort would be to define a library with often used elements of KADS models.

7 Conclusion

In this paper we have described a language for formally representing KADS models of expertise. It turned out to be possible to represent all of the components of a model of expertise in a language that is a combination of a number of logical constructs: order-sorted first order predicate calculus, meta-logic and dynamic logic, each of which is well understood, has known properties, a well-defined proof theory, and, perhaps most important of all, a clear declarative semantics. We feel that KADS models of expertise can be adequately expressed in (ML)² according to the guidelines given in this paper. Furthermore, in the process of defining the (ML)² constructs that represent KADS notions, a number of these KADS notions were further clarified (e.g. the nature of the connections between the layers) and refined (e.g. the distinction between types of meta-classes).

(ML)² is not the only attempt at formalising KADS models of expertise. However, (ML)² differs from some of the other approaches because (ML)² models are meant as a *formalisation* of models of expertise rather than as a way to *mechanise* them. For instance, the MODEL-K approach from (Karbach *et al.*, 1990) is mainly aimed at mechanising a model of expertise, and not at providing a declarative representation. As a result, MODEL-K representations can contain arbitrary pieces of code, which do not lend themselves very well for inspection, derivation, etc. Some other approaches are perhaps closer in spirit to (ML)², notably (Wetter, 1990) and KARL (Angele, Fensel, Landes & Studer, 1991; Fensel, Angele & Landes, 1991). We would argue that the formalism presented in [Wetter, 1990] contains a number of constructions that can only be understood through a procedural (i.e. non-declarative) semantics. The KARL language more closely resembles (ML)², since it uses a purely declarative language for domain and inference layer. A major difference is that KARL is restricted to function-free Horn logic for representing domain and inference layers. It is an open question whether this restriction (not made in (ML)²) is not too strong. Secondly, KARL uses a more conventional procedural language at the task layer than the dynamic logic employed by (ML)². Finally, DESIRE by (Treur, 1991) also shares a number of properties with (ML)², notably the use of meta-constructions as a way of capturing the relation between different layers in a model, but the DESIRE language has no strong underlying conceptual model, in the way that (ML)² and others are based on KADS.

Our practical experience with using (ML)² on real-life models of expertise is encouraging. We gain the advantages of formal representations over informal ones (removal of ambiguity, validation). On the other hand, constructing a formal model is quite a complex activity and even formalisations of simple models of expertise are of a significant size.

Acknowledgements

We are grateful for many useful comments on earlier versions of this paper from Manfred Aben, Hans Akkermans (who also suggested the use of dynamic logic), Brigitte Bartsch-Spörl, Gertjan van Heijst, Guus Schreiber, Annette ten Teije, Peter Terpstra, Angi Voß, Bob Wielinga and two referees from this journal.

References

- [Akkermans et al., 1990] Akkermans, H., van Harmelen, F., Schreiber, G., and Wielinga, B. (1990). A formalisation of knowledge-level models for knowledge acquisition. *International Journal of Intelligent Systems*. Forthcoming.
- [Angele et al., 1991] Angele, J., Fensel, D., Landes, D., and Studer, R. (1991). Sisyphus - no problem with KARL. In *Proceedings of the European Knowledge Acquisition Workshop EKAW'91, (Sisyphus Working Papers Part 2)*, Strathclyde.
- [Balder, 1991] Balder, J. (1991). TheME: a Theory Manipulation Environment for creating Formal Knowledge Models. ESPRIT Project P5248 KADS-II/T1.2/TR/ECN/002/1.0, Netherlands Energy Research Centre ECN.
- [Bergstra et al., 1990] Bergstra, J., Heering, J., and Klint, P. (1990). Module algebra. *Journal of the ACM*, 37(2):335–372.
- [Brachman and Schmolze, 1985] Brachman, R. and Schmolze, J. (1985). An overview of the KL-ONE knowledge representation system. *Cognitive Science*, 9:171–216.
- [Bredeweg et al., 1990] Bredeweg, B., Reinders, M., and Wielinga, B. (1990). Garp: A unified approach to qualitative reasoning. Technical Report VF-memo 117, University of Amsterdam.
- [Breuker et al., 1987] Breuker, J., Wielinga, B., van Someren, M., de Hoog, R., Schreiber, G., de Greef, P., Bredeweg, B., Wielemaker, J., Billault, J.-P., Davoodi, M., and Hayward, S. (1987). Model Driven Knowledge Acquisition: Interpretation Models. ESPRIT Project P1098 Deliverable D1 (task A1), University of Amsterdam and STL Ltd.
- [Brogi et al., 1990] Brogi, A., Mancarella, P., Pedreschi, D., and Turini, F. (1990). Hierarchies through basic meta-level operators. In Bruynooghe, M., editor, *Proceedings of the Second Workshop on Meta-programming in Logic (META '90)*, pages 381–396, Leuven, Belgium.
- [Chandrasekaran, 1987] Chandrasekaran, B. (1987). Towards a functional architecture for intelligence based on generic information processing tasks. In *Proceedings of the 10th IJCAI*, pages 1183–1192, Milano.
- [Cohn, 1985] Cohn, A. (1985). On the solution of Schubert's steamroller in many sorted logic. In *Proceedings of the Ninth IJCAI*, pages 345–352, Los Angeles. International Joint Conference on Artificial Intelligence.
- [Eshelman, 1989] Eshelman, L. (1989). MOLE: A knowledge-acquisition tool for cover-and-differentiate systems. In Marcus, S., editor, *Automating Knowledge Acquisition for Expert Systems*, pages 37–80. Kluwer Academic Publishers, The Netherlands.
- [Feferman, 1962] Feferman, S. (1962). Transfinite Recursive Progressions of Axiomatic Theories. *Journal of Symbolic Logic*, 27(3):259–316.

- [Fensel et al., 1991] Fensel, D., Angele, J., and Landes, D. (1991). Knowledge representation and acquisition language (KARL). In *Proceedings 11th International workshop on expert systems and their applications (Volume: Tools and Techniques)*, Avignon.
- [Giunchiglia and Smaill, 1988] Giunchiglia, F. and Smaill, A. (1988). Reflection in constructive and non-constructive automated reasoning. In Abramson, H. and Rogers, M., editors, *Meta-Programming in Logic Programming (META '88)*, pages 123–140, Bristol. MIT Press. Also: DAI Research Paper 375, Dept. of Artificial Intelligence, Edinburgh.
- [Harel, 1984] Harel, D. (1984). Dynamic logic. In Gabbay, D. and Guenther, F., editors, *Handbook of Philosophical Logic, Vol. II: extensions of Classical Logic*, pages 497–604. Reidel Publishing Company, Dordrecht, The Netherlands.
- [Hayes, 1977] Hayes, P. (1977). In defence of logic. In *Proceedings of IJCAI-77*, pages 559–556. International Joint Conference on Artificial Intelligence.
- [Jackson et al., 1989] Jackson, P., Reichgelt, H., and van Harmelen, F. (1989). *Logic-Based Knowledge Representation*. The MIT Press, Cambridge, MA.
- [Karbach et al., 1990] Karbach, W., Linster, M., and Voß, A. (1990). Model-based approaches: One label - one idea? In Wielinga, B., Boose, J., Gaines, B., Schreiber, G., and van Someren, M., editors, *Current Trends in Knowledge Acquisition*, pages 173–189. IOS Press, Amsterdam.
- [Karbach et al., 1991] Karbach, W., Voß, A., Schukey, R., and Drouwen, U. (1991). Model-K: Prototyping at the knowledge level. In *Proceedings Expert Systems '91, Avignon, France*, pages 501–512.
- [Kowalski, 1979] Kowalski, B. (1979). *Logic for Problem Solving*. Artificial Intelligence Series. North-Holland Publisher, Amsterdam.
- [Maes and Nardi, 1988] Maes, P. and Nardi, D., editors (1988). *Meta-Level Architectures and Reflection*, Amsterdam. North-Holland.
- [McDermott, 1989] McDermott, J. (1989). Preliminary steps towards a taxonomy of problem-solving methods. In Marcus, S., editor, *Automating Knowledge Acquisition for Expert Systems*, pages 225–255. Kluwer Academic Publishers, The Netherlands.
- [Moore, 1982] Moore, R. (1982). The role of logic in knowledge representation and common-sense reasoning. In *National Conference on Artificial Intelligence*, pages 428–433, Pittsburgh, PA. American Association for Artificial Intelligence.
- [Moore, 1984] Moore, R. (1984). The role of logic in artificial intelligence. Note 335, SRI International.
- [Musen, 1989] Musen, M. (1989). *Automated Generation of Model-Based Knowledge-Acquisition Tools*. Pitman, London. Research Notes in Artificial Intelligence.
- [Reinders et al., 1991] Reinders, M., Vinkhuyzen, E., Voß, A., Akkermans, H., Balder, J., Bartsch-Sporl, B., Bredeweg, B., Drouwen, U., van Harmelen, F., Karbach, W., Karssen, Z., Schreiber, G., and Wielinga, B. (1991). A conceptual modelling framework for knowledge-level reflection. ESPRIT Basic Research Action P3178 REFLECT RFL/UvA/M/4, University of Amsterdam. Submitted to AICOM.
- [Sanella and Burstall, 1983] Sanella, D. and Burstall, R. (1983). Structured theories in LCF. In *Proceedings of the 8th Colloquium on Trees in Algebra and Computing*. Springer Verlag. Lecture Notes in Computer Science No. 159.
- [Schmidt-Schauß, 1989] Schmidt-Schauß, M. (1989). *Computational Aspects of an Order-Sorted Logic with Term Declarations*. Springer-Verlag, Berlin. Lecture Notes in Artificial Intelligence No. 395.
- [Schreiber et al., 1990] Schreiber, G., Wielinga, B., and Akkermans, H. (1990). Differentiating problem solving methods. ESPRIT Basic Research Action P3178 REFLECT, Technical Report RFL/UvA/I.4/6, University of Amsterdam.
- [Smith, 1984] Smith, B. (1984). Reflection and semantics in Lisp. In *Proc. 11th ACM Symposium on Principles of Programming Languages*, pages 23–35, Salt Lake City, Utah. also: Xerox PARC Intelligent Systems Laboratory Technical Report ISL-5.
- [Tarski, 1936] Tarski, A. (1936). Der Wahrheitsbegriff in den formalisierten Sprachen. *Studia Philosophica*, 1:261–405. English translation in *Logic, Semantics, Metamathematics*, A. Tarski, Oxford University Press, 1956.

- [ten Teije et al., 1991] ten Teije, A., van Harmelen, F., and Reinders, M. (1991). Si(ML)²: a prototype interpreter for a subset of (ML)². Technical Report ESPRIT Project P5248 KADS-II/T1.2/TR/UvA/005/1.0, University of Amsterdam & Netherlands Energy Research Foundation ECN.
- [Treur, 1991] Treur, J. (1991). On the use of reflection principles in modelling complex reasoning. *International Journal of Intelligent Systems*, 6.
- [van Harmelen, 1991] van Harmelen, F. (1991). *Meta-level Inference Systems*. Research Notes in AI. Pitmann, Morgan Kaufmann, London, San Mateo California.
- [van Harmelen et al., 1990] van Harmelen, F., Akkermans, H., Balder, J., Schreiber, G., and Wielinga, B. (1990). Formal specifications of knowledge models. ESPRIT Basic Research Action P3178 REFLECT, Technical Report RFL/ECN/I.4/1, Netherlands Energy Research Foundation ECN.
- [van Harmelen et al., 1991] van Harmelen, F., Balder, J., Aben, M., and Akkermans, H. (1991). (ML)²: A formal language for kads conceptual models. Technical Report ESPRIT Project P5248 KADS-II/T1.2/TR/ECN/006/1.0, Netherlands Energy Research Foundation ECN & University of Amsterdam. Deliverable D1.2.1. A shorter version is published in *Knowledge Acquisition Journal*, 1992.
- [Walther, 1984] Walther, C. (1984). A mechanical solution of Schubert's steamroller by many-sorted resolution. In *Proceedings of the fourth National Conference on Artificial Intelligence*, pages 330–334, Austin, Texas. American Association for Artificial Intelligence.
- [Walther, 1987] Walther, C. (1987). *A many-sorted calculus based on resolution and paramodulation*. Research Notes in Artificial Intelligence. Pitman, London.
- [Wetter, 1990] Wetter, T. (1990). First-order logic foundation of the KADS conceptual model. In Wielinga, B., Boose, J., Gaines, B., Schreiber, G., and van Someren, M., editors, *Current trends in knowledge acquisition*, pages 356–375, Amsterdam. IOS Press.
- [Weyhrauch, 1980] Weyhrauch, R. (1980). Prolegomena to a theory of mechanized formal reasoning. *Artificial Intelligence*, 13. Also in: *Readings in Artificial Intelligence*, Webber, B.L. and Nilsson, N.J. (eds.), Tioga publishing, Palo Alto, CA, 1981, pp. 173-191. Also in: *Readings in Knowledge Representation*, Brachman, R.J. and Levesque, H.J. (eds.), Morgan Kaufman, California, 1985, pp. 309-328.
- [Wielinga et al., 1991] Wielinga, B., Schreiber, A., and Breuker, J. (1991). KADS: A modelling approach to knowledge engineering. ESPRIT Project P5248 KADS-II/T1.1/PP/UvA/008/1.0, University of Amsterdam. Submitted to *Knowledge Acquisition*.
- [Wielinga et al., 1992] Wielinga, B., Schreiber, A., and Breuker, J. (1992). KADS: A modelling approach to knowledge engineering. *Knowledge Acquisition*. this issue. Also as: Technical Report ESPRIT Project P5248 KADS-II/T1.1/PP/UvA/008/1.0, University of Amsterdam.

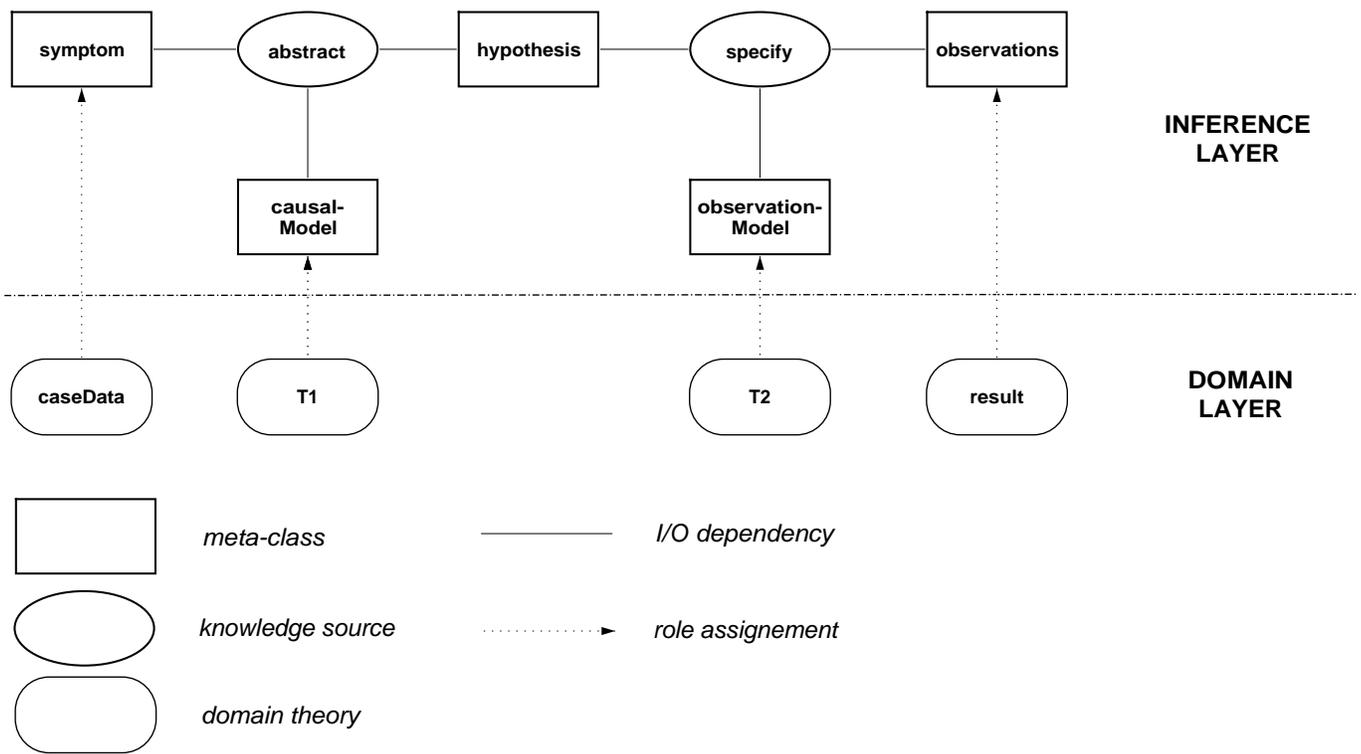


Figure 3: Inference structure for example

topology	type	axiom
input, not output	initial	7
input and output	internal	8
output, not input	terminal	9

Figure 4: Types of meta-classes derived from their input/output characteristics