Proceedings of the Tenth European Conference on AI (ECAI'92), pp. 582-586

$(ML)^2$: A formal language for KADS models (short version)

Frank van Harmelen*, John Balder**

SWI, University of Amsterdam, Roetersstraat 15, NL-1018 WB Amsterdam, e-mail: frankh@swi.psy.uva.nl

Abstract. We present $(ML)^2$, a formal language for the representation of KADS models of expertise. $(ML)^2$ is a combination of first order predicate logic (for the declarative representation of domain knowledge), meta-logic (for the representation of how to use the domain knowledge) and dynamic logic (for the representation of control information). After a brief summary of KADS, we describe how each of the four KADS layers is represented in $(ML)^2$, and we compare our formalism to other formalisms that have been proposed for the formalisation of KADS models.

1 Introduction

One of the central concerns of "knowledge engineering" is the construction of a model of problem solving behaviour. One of the prominent approaches in recent years to this problem (at least in Europe) has been the KADS methodology for knowledge engineering [9]. KADS is centered around a so-called *model of expertise* which describes the problem solving expertise of the system to be modelled independent of a possible implementation.

Traditionally, these models have always been expressed in an informal way, using a vocabulary of natural language, semi-structured language and graphical notation. In this paper, we present $(ML)^2$, a formal language for the representation of KADS models. This paper is a short version of a more detailed presentation of the language [7], and is intended as a description of the language for a wider audience.

This paper is structured as follows: to keep this paper self-contained, we first give a brief description of KADS models (section 2). We present $(ML)^2$ by showing how it represents each of the four layers of a KADS model (sections 3-7). Finally, we compare $(ML)^2$ with some other recent proposals for formalising KADS models (section 8).

*This work is part of research projects partially funded by the ESPRIT Programme of the Commission of the European Communities as project numbers 3178 (REFLECT) and 5248 (KADS-II).

**Netherlands Energy Research Foundation ECN Petten, The Netherlands

2 A brief description of KADS models

A central feature of the KADS methodology for constructing knowledge-based systems is the so-called model of expertise. Its goal is to provide a model of the problem solving behaviour required of the knowledge-based system in an implementation independent way. KADS models consist of four hierarchically organised layers and prescribe the contents of the layers and the relations among them, as follows:

Domain layer: This is the "lowest" of the four layers, and represents knowledge about the application domain of the system. An important property of the domain layer is that the knowledge should be represented as much as possible independently from the way it will be used (i.e. the domain layer is a *declarative representation* of the domain knowledge of a system).

Inference layer: This second layer plays a central role. It specifies how to *use* the knowledge from the domain layer. This is done in two ways: the inference layer specifies (1) the *basic inference steps* that can be made using the domain knowledge (these basic inference steps are known as "knowledge sources"), and (2) the *roles* that the elements of the domain knowledge can play in the inference process. These roles are known as "meta-classes". The inference layer also specifies the data-dependencies between these steps and roles. The inference layer does *not* specify any control knowledge: no ordering is imposed on the various inference steps.

Task layer: The purpose of the task layer is to specify *control* over the execution of the basic inference steps specified at the inference layer. It does this by imposing an ordering on these steps in terms of execution sequences, iterations, conditional statements etc.

Strategy layer: This "highest" of the four levels in a KADS models is concerned with *task selection*: how to choose between various tasks that achieve the same goal.

For a more detailed description of KADS, we refer to [9].

3 The domain layer in (ML)^2

The domain layer represents declarative knowledge about the domain of application: *Logic* has been developed to represent exactly this kind of information, and it is therefore not surprising that we chose first order predicate logic, as the representation language for the domain layer.

For practical reasons, we include two extensions to the language of first order logic: we use *order-sorted logic* because it is more compact and combinatorially tractable than unsorted logic, and we divide our axioms into *sub-theories* to give us a mechanism for modularisation. Both these extensions are *conservative* in the sense that they do not alter the strength of the logic: they are only notational devices.

No other aspects of (ML)^2 depend on the fact that we use first order predicate logic on the domain layer, and if required by the application, we can easily extend (ML)^2 to use temporal, modal or other non-standard logics.

A domain theory in (ML)^2 consists of the declaration of the language of the theory (the signature), plus the axioms of the theory. A simple example is given below:

theory T_1
signature
  sorts reading, car ;
  constants myCar:car ;
  functions gasDial:car $\rightarrow$ reading ;
  predicates
    engineDoesntRun: car ;
    noGas:reading ;
axioms
  $\forall$ X:car[noGas(gasDial(X)) $\rightarrow$ engineDoesntRun(X)] ;
endtheory

4 The inference layer in (ML)^2

The purpose of the inference layer is to state what the potential *inference steps* (knowledge sources) are that can be made using knowledge from the domain layer, and what *roles* the various domain expressions will play in these steps. In other words, the inference layer is a theory *about* the domain layer, namely about the use of the domain layer. This makes the inference layer a *meta*-layer of the domain layer, in the technical sense of *meta-*: a theory $\mathcal{M}$ is a meta-theory of a theory $\mathcal{O}$ if (some of the) terms from $\mathcal{M}$ refer to formula from $\mathcal{O}$.

Representing meta-classes: In any meta-logic, the meta-theory must have *names* for the expressions from the object-theory in order to refer to these object-expressions. In (ML)^2 we exploit these names to encode the *roles* that the object-expressions play in the inference process (the KADS *meta-classes*). Since knowledge-engineers decide which meta-classes feature in a KADS model, the knowledge-engineers must be able to *define* the names of domain-expressions. In order to encode these meta-classes, it must be possible to give different names (for different meta-classes) to syntactically similar expressions. This departs from standard constructions in meta-logic where the meta-names of object-expressions depend only on the syntactic form of the expressions.

To achieve definable names, we allow the knowledge engineer to specify sets of *rewrite rules*. Such a set of rewrite rules defines how a domain-expression must be "rewritten" to obtain its meta-name. Such a set of rewrite rules is called a *lift-definition* in (ML)^2. A lift-definition also defines (through a signature definition) the language-elements in the meta-theory that are used to represent the meta-classes. Typically, for any meta-class m, we introduce a function symbol $m()$ in the meta-theory. For example[1]

lift-definition cause&abstract **from** T_1, T_2 ;
signature
  constants "P_1", "P_2" ;
  functions
    causation, cause, symptom,
    abstraction, element, class ;
lift-variables P_1, P_2:predicate ;
mapping
  lift(T_1, $P_1 \rightarrow P_2$) $\mapsto$
    causation(cause("P_1"),symptom("P_2"));
  lift(T_2, $P_1 \rightarrow P_2$) $\mapsto$
    abstraction(element("P_1"),class("P_2"));
end-lift-definition

This lift-definition introduces the meta-classes *causation* and *abstraction*, and define that implications from theory T_1 will be interpreted as *causation*s (mapping *cause*s to *symptom*s). Similar looking implications, but from T_2, will be interpreted as *abstraction*s (mapping concrete *element*s to abstract *class*es).

Representing knowledge sources: The second aspect of an inference layer are the primitive inference steps (knowledge sources). Such knowledge sources map a number of input meta-classes onto a single output meta-class. In (ML)^2, knowledge sources are represented by meta-level theories of a restricted form. A knowledge source KS_k corresponds to a theory with axioms of the form

$$LHS_{KS_k} \rightarrow KS_k(t_1, \ldots, t_n, t_{n+1}) \qquad (1)$$

(or any formulation that is logically equivalent to this). The left-hand side LHS_{KS_k} can be an arbitrary formula constructed from reflective predicates and predicates of the form $input_{MC_i}(t_i)$, and each t_i will be a term whose outermost function symbol represents

[1] for simplification, we have left out the type declarations from this section

the meta-class MC_i, along the lines defined above. We will postpone the definition of the $input_{MC_i}$ predicates to section 5. We call the predicate KS_k the *knowledge source predicate*. Such a knowledge source predicate, axiomatised by formulae of form (1), represents the knowledge source as an $(n+1)$-place relation between the n input meta-classes and the single output meta-class.

A very simple example of knowledge source definition is:

theory abduct
use causes, symptoms;
signature
  predicates $\mathrm{KS}_{\mathrm{abduct}}$;
variables X, Y ;
axioms
  $\forall$X[$\mathrm{input}_{\mathrm{causes}}$(causation(cause(X), symptom(Y))) $\wedge$
    $\mathrm{input}_{\mathrm{symptoms}}$(symptom(Y)) $\rightarrow$
      $\mathrm{KS}_{\mathrm{abduct}}$(symptom(Y),
        causation(cause(X), symptom(Y)),
        cause(X))]
endtheory

As can be seen from this example, a knowledge source definition can **use** lift-definitions, which results in the signature of the lift-definition becoming available to the knowledge source theory.

Reflection rules in $(\mathrm{ML})^2$**:** Besides the naming relation defined by the lift-operators, there is an additional connection between inference- and domain-layer (or: between meta- and object-theory), namely through the use of inference rules that provide a link between inference in the two layers. In $(\mathrm{ML})^2$, we require three inference rules between meta- and object-layers, generally known as *reflection rules*:

$$\frac{\vdash_{\mathcal{O}} \phi}{\vdash_{\mathcal{M}} ask^{\vdash}(\mathcal{O}, \lceil\phi\rceil)} \ (up); \quad \frac{\vdash_{\mathcal{M}} tell(\lceil\mathcal{O}\rceil, \lceil\phi\rceil)}{\vdash_{\mathcal{O}} \phi} \ (down);$$

$$\frac{\phi \in \mathcal{O}}{\vdash_{\mathcal{M}} ask^{\in}(\lceil\mathcal{O}\rceil, \lceil\phi\rceil)} \ (axiom)$$

(where the meta-term $\lceil\phi\rceil$ is the name for the object-formula ϕ, as defined through lift-definitions). Rule (up) states that if a formula ϕ is provable in the object-theory $\mathcal{O}$, then the formula $ask^{\vdash}(\lceil\mathcal{O}\rceil, \lceil\phi\rceil)$ is provable in the meta-theory $\mathcal{M}$, allowing inferences in $\mathcal{O}$ to affect inferences in $\mathcal{M}$. Conversely, rule $(down)$ allows inferences in $\mathcal{M}$ to affect inferences in $\mathcal{O}$. Finally, rule $(axiom)$ states that if formula ϕ is an axiom of $\mathcal{O}$, then $ask^{\in}(\lceil\mathcal{O}\rceil, \lceil\phi\rceil)$ is provable in $\mathcal{M}$.

5 The task layer in $(\mathrm{ML})^2$

The purpose of the task-layer in a KADS model is to enforce *control* over the inference steps specified at the inference layer.

In $(\mathrm{ML})^2$ we employ Quantified Dynamic Logic (QDL) to represent the task layer. QDL is a modal extension of first order logic developed by computer scientists for reasoning about properties of programs [3]. Before describing the use of QDL in $(\mathrm{ML})^2$ task layers, we first give a brief introduction to QDL.

Quantified Dynamic Logic: In QDL, first order logic is extended with the notions of program, variable and state. 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. A program operates on an *execution state*, determined by the current value of all its variables. A *program* is conceived as a transformation from its *initial state* into its *final state*. QDL introduces a single type of atomic program, the *assignment statement* $x := t$ (with x a variable and t a term) which maps any state into a similar state but with variable x having the new value t. Three *program constructors* allow the composition of complex programs out of atomic ones: if α and β are programs and ϕ is a predicate, then the following are also programs: $\alpha;\beta$ (do α followed by β); $\alpha \bigcup \beta$ (do either α or β, nondeterministically); $\alpha\star$ (repeat α a nondeterministic finite number of times); $\phi?$ (proceed if ϕ is true, else fail). These elementary constructs allow the definition of various traditional programming constructs such as *if-then-else*, *while-do*, etc.

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: *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*.

The *semantics* of dynamic logic is a modal one, where a "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 such that s can be reached by executing α, and ϕ is true in state s.

Tasks as programs: We now explain how we exploit the machinery of QDL to represent the task layer of a KADS model. Since the purpose of a task layer is to enforce control over the inference layer, it is natural to represent the task layer as a QDL *program*, which expresses how the knowledge sources from the inference layer should be "executed". QDL's test-operator "?" allows us to turn the declarative representation of a knowledge source (as the $(n+1)$-place relation KS_k from formula (1)) into a program that can be "called" from the task-layer.

Representing states: Since at the task layer we want to "execute" knowledge sources, we require a representation of the *state* of the inference process. We use QDL variables for this purpose as follows: for

each knowledge source KS_i, we assume a QDL variable V_{KS_i} whose value will be a tuple of all input/output relations that have been computed so far for knowledge source KS_i.

Furthermore, for each meta-class MC_j we assume a QDL variable V_{MC_j} whose value will be the tuple of all values that have been computed for meta-class MC_j.

The entire state of an inference process is now represented by the collection of all variables V_{KS_i} and V_{MC_j} (one variable for every knowledge source and for every meta-class).

Primitive operations on knowledge sources: The above representation of the state of the inference process allows us to define the following four primitive operations on any knowledge source $KS_i(\vec{I}, O)$ (we write $\vec{I}$ as an abbreviation for a sequence of variables $I_1, \ldots, I_n$):

- *has-solution-*$KS_i(\vec{I}, O)$ is true *iff* the tuple $\langle \vec{I}, O \rangle$ satisfies the knowledge source predicate KS_i. This operation is independent of the current state of the inference process.
- *old-solution-*$KS_i(\vec{I}, O)$ is true *iff* the tuple $\langle \vec{I}, O \rangle$ has previously been computed as the result of "executing" KS_i. The knowledge source variable KS_i is inspected for this purpose.
- *more-solutions-*$KS_i(\vec{I}, O)$ is true *iff* the tuple $\langle \vec{I}, O \rangle$ is a previously uncomputed solution to KS_i. This can be defined in terms of the previous two predicates.
- *give-solution-*$KS_i(\vec{I}, O)$ is true *iff* the tuple $\langle \vec{I}, O \rangle$ is a previously uncomputed solution, but the new solution will also be recorded in the state of the inference process. This operation corresponds to "calling" a knowledge source from the task layer and storing the result in the process state, whereas the other three operations do not alter the state of the computations. Consequently, the other 3 operations are *predicates* of QDL, and *give-solution-*KS_i is the only operation that corresponds to a *program* in QDL.

Notice that the execution of this program does not specify in any way in which order the different solutions to KS_i will be computed. This is in accordance with the principle in KADS that knowledge sources are computional units that do not require any further internal control.

Using these four basic operations, we are now in a position to define a task: a task in a formalised KADS model is a QDL program defined out of the expressions *has-solution-*KS_i, *old-solution-*KS_i, *more-solutions-*KS_i and *give-solution-*KS_i (for each knowledge source KS_i).

Using the semantics of QDL, we see that a task in $(\mathrm{ML})^2$ is a program that maps one state of the inference process onto another state, with states represented by the collection of variables V_{KS_i} and V_{MC_j}.

The *input* predicates: In section 4, we used predicates of the form $input_{MC_i}(t_i)$ in the axioms for the knowledge source predicates. These predicates represent the input meta-classes MC_i to the knowledge source. In $(\mathrm{ML})^2$, the contents of a meta-class can be obtained in two ways: since meta-classes are descriptions of (the role of) domain expressions, we can retrieve the contents of meta-classes by referring to the contents of domain theories. In this case, the $input_{MC_i}$ predicate can be defined as

$$\forall x : input_{MC_i}(x) \leftrightarrow ask^{\in}(O, x) \qquad (2)$$

where O is (the name of) the object-theory mentioned in the left-hand side of the rewrite rules in the lift-operator for meta-class MC_i[2].

Alternatively, we can retrieve the contents of meta-classes from the V_{MC_j} variables used to store the state of the inference process, by using one of the following:

$$\forall x : input_{MC_j}(x) \quad \leftrightarrow \quad \exists y : V_{MC_j} = \langle x | \ldots \rangle \qquad (3)$$

$$\forall x : input_{MC_j}(x) \quad \leftrightarrow \quad x \in V_{MC_j} \qquad (4)$$

$$\forall x : input_{MC_j}(x) \quad \leftrightarrow \quad x = V_{MC_j} \qquad (5)$$

$$(6)$$

We use (3) if we are interested in the most recently computed value, (4) if we are interested in any previously computed value, or (5) if we want all previously computed values. Thus, our formalism allows for any of the multiple uses that are often made of the contents of meta-classes in KADS models, but forces the user to make clear in which way each meta-class is used.

6 The strategy layer in $(ML)^2$

Although the strategy layer is the least well developed layer of KADS models, it is generally perceived as *task-selection*: given various tasks for achieving various goals, which task should be chosen under which circumstances?

The language of QDL incorporated in $(\mathrm{ML})^2$ provides a natural way to represent such information: an expression of the form $\phi \rightarrow [\tau]\psi$ can be interpreted as: "given certain preconditions ϕ, program τ is a way of achieving ψ". Expressions of this form can be used to derive 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 τ_1, τ_2 and τ_3:

$$\begin{array}{rcl} \phi_1 & \rightarrow & [\tau_1]\phi_2 \\ \phi_3 & \rightarrow & [\tau_2]\phi_4 \\ \phi_2 \vee \phi_4 & \rightarrow & [\tau_3]\phi_5 \end{array}$$

[2]Formula (2) might suggest that $ask^{\in}$ is the only predicate used in formulating KADS models in $(\mathrm{ML})^2$. However, the reader should remember that other reflective predicates, notably $ask^{\vdash}$ can occur in the bodies of knowledge source predicates, as specified in section 4.

we can deduce that the program $(\phi_1?;(\tau_1;\tau_3)) \cup (\phi_3?;(\tau_2;\tau_3))$ is a way of achieving goal ϕ_5.

7 Relation between the layers

Although an earlier publication on (ML)^2 [1] presented inference, task and strategy layer each as a meta-layer of the layer below, the current relation between the layers in (ML)^2 is much more diverse. As described above, the relation between domain and inference layer is an *object/meta-relation*. The relation between inference and task layer on the other hand is entirely different: the inference layer (a set of first order theories) is *embedded* in the task layer (a QDL theory, containing first order logic as a subset). The relation between task and strategy layer is different again: both are theories in QDL, but the strategy layer *extends* the task layer with additional axioms that comprise the strategic knowledge concerning properties of tasks.

8 Comparison and conclusions

(ML)^2 is not the only attempt at formalising KADS models. 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 [5] is mainly aimed at mechanising a model, 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 to inspection, derivation, etc.

Some other approaches are perhaps closer in spirit to (ML)^2, notably FORKADS [8], KARL [2] VITAL-CML [4], and DESIRE [6]. A major drawback of FORKADS is that it provides no syntactic distinction between domain and inference layers, and as such does not *force* the formal model to have the form required by KADS in the same way that other formalisms (including (ML)^2) do.

KARL resembles (ML)^2 in many respects, but it is restricted to function-free Horn logic for representing domain and inference layers. It is an open question whether this restriction (made with an eye to mechanising KARL models, and absent from (ML)^2) is not too strong.

The VITAL-CML language is also close in spirit to (ML)^2, particularly in its use of modularised first order theories. It employs parameterised theories as a very elegant way of connecting domain layer and inference layer, and the relation between this solution and the one chosen in (ML)^2 (an object/meta-construction) deserves further study.

Finally, DESIRE 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.

Conclusions: We have presented (ML)^2, a formal language for representing KADS models. 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. (ML)^2 can be summarised by the following pseudo-equation

$$\text{(ML)}^2 = \text{FOPC} + \text{sorts} + \text{sub-theories} + \text{meta-logic} + \text{QDL}$$

These components of (ML)^2 have been motivated as follows: (1) Logic is used at the domain layer because it is well suited for the declarative representation of knowledge independent of use. Sorts and sub-theories are simply pragmatic conservative extensions. (2) Meta-logic is used to represent the inference layer since the inference layer is *about* the use of the knowledge at the domain layer. (3) QDL is used to represent the task layer, since this layer is represents procedural knowledge (sequence, state) and QDL is one of the few formalisms that offer a declarative representation of this type of knowledge.

Each of these components is well understood, and has known properties, a well-defined proof-theory and a clear declarative semantics.

References

[1] H. Akkermans, F. van Harmelen, G. Schreiber, and B. Wielinga. A formalisation of knowledge-level models for knowledge acquistion. *International Journal of Intelligent Systems*, 1991. forthcoming.

[2] J. Angele, D. Fensel, D. Landes, and R. Studer. KARL: An executable language for the conceptual model. In *6th Workshop on Knowledge Acquisition*, pages 1.1–20, Banff, 1991.

[3] D. Harel. Dynamic logic. In D. Gabbay and F. Guenthner, editors, *Handbook of Philosophical Logic, Vol. II: extensions of Classical Logic*, pages 497–604. 1984.

[4] W. Jonker and J.W. Spee. Yet another formalisation of KADS conceptual models. In Th. Wetter, editor, *EKAW*, 1992.

[5] W. Karbach, A. Voß, R. Schukey, and U. Drouwen. Model-K: Prototyping at the knowledge level. In *Avignon*, pages 501–512, 1991.

[6] J. Treur. On the use of reflection principles in modelling complex reasoning. *International Journal of Intelligent Systems*, 6, 1991.

[7] F. van Harmelen and J. Balder. (ML)^2: a formal language for KADS models of expertise. *Knowledge Acquisition*, 4(1), 1992.

[8] T. Wetter. First-order logic foundation of the KADS conceptual model. In B. Wielinga et al., editor, *Current trends in knowledge acquisition*, pages 356–375, 1990.

[9] B. J. Wielinga, A. Th. Schreiber, and J. A. Breuker. KADS: A modelling approach to knowledge engineering. *Knowledge Acquisition*, 4(1), 1992.