A KADS/(ML)$^2$ Model of a Scheduling Task

John Balder  
Netherlands Energy Research Foundation ECN  
Informatics SE&R  
P.O.box 1, NL-1755ZG Petten  
The Netherlands  
Email: balder@ecn.nl

Frank van Harmelen  
Manfred Aben  
University of Amsterdam  
Department of Social Science Informatics  
Roeterstraat 15, NL-1018WB Amsterdam  
The Netherlands  
Email: {frankh.manfred}@swi.psy.uva.nl

Part I: Description of the (ML)$^2$ language

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 [15]. 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 KADS models have always been expressed in an informal way, using a vocabulary of natural language, semi-structured language and graphical notation. In recent years, we have developed (ML)$^2$, a more formal language to express KADS models. In this paper, we first give a brief overview of the (ML)$^2$ language, and then show how this language can be used to model the simple scheduling task.

Part I of this paper is devoted to a description of (ML)$^2$. To make this paper self-contained, we give a brief description of KADS models (Section 2), and then describe how (ML)$^2$ captures each of the elements of a KADS model (Sections 3–5), followed by a concluding section (6). Part II of this paper shows how (ML)$^2$ can be used to model the example scheduling task [9]. After a brief introduction (Section 7), we again treat each of the KADS layers in turn (Sections 8 – 10).

Part I of this paper is based on [4]. A more extensive description of (ML)$^2$ can be found in [5].

---

6This work is part of research projects partially funded by the ESPRIT Programme of the Commission of the European Communities as project numbers 9178 (REFLECT) and 5248 (KADS-II).
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 behavior 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 task-neutral and *declarative representation* of the domain knowledge of a system). Thus, the domain layer is represented independent from how it will be used.

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 “primitive inference actions” or PIA’s\(^1\)), and (2) the *roles* that the elements of the domain knowledge can play in the inference process. These roles are known as “dynamic knowledge roles” or DKR’s\(^2\). The inference layer specifies the basic inference steps and knowledge roles, and also specifies the data-dependencies between these steps and roles. These dependencies are specified in a network of inference steps and knowledge roles known as an *inference structure*. Although the inference layer specifies the basic inference steps, it does not specify any control knowledge: no ordering is imposed on the various inference steps. Thus, the inference layer is represented independent from when the steps will be made.

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. In this paper we do not deal with the strategy layer at all, but see [4] and [5] for speculations on how this layer could be captured in ML\(^2\).

\(^1\)They used to be called “knowledge-sources”.
\(^2\)Previously known as “meta-classes”.

For a more detailed description of KADS, we refer to [15].
3 The domain layer in \( (\text{ML})^2 \)

The domain layer represents declarative knowledge about the domain of application: facts and rules that are true in the domain, represented independently from how they are going to be used. Logic has been developed to represent exactly this kind of information, and it is therefore not surprising that we chose logic, and in particular 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. First of all, we use order-sorted logic, since it is well established that the introduction of sorts and subsorts reduces the number of axioms, simplifies the remaining axioms, shortens the lengths of proofs, and makes theories easier to read.

Secondly, if \((\text{ML})^2\) 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. 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 \((\text{ML})^2\), we currently use union of theories, but more sophisticated operators for combining theories have been proposed in the literature. These more sophisticated operators also allow for automatic resolving of name-clashes, while the simple union operator provided in \((\text{ML})^2\) does not allow for this.

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 \((\text{ML})^2\) 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 \((\text{ML})^2\) to use temporal, modal or other non-standard logics.

A domain theory in \((\text{ML})^2\) consists of the declaration of the language of the theory (the signature), plus the axioms of the theory. In \((\text{ML})^2\) we will also freely use mathematical structures often needed for modelling knowledge in various domains, such as numbers, sets, tuples, trees, etc.

We refer to Section 8 for some simple examples of domain theories.

4 The inference layer in \( (\text{ML})^2 \)

The purpose of the inference layer is to state what the potential inference steps (PIA's) 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 formulae from \( \mathcal{O} \).

Representing knowledge roles: 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 \((\text{ML})^2\) we exploit these names to encode the roles that the object-expressions play in the inference process. Since knowledge-engineers decide which knowledge roles feature in a KADS model, the knowledge-engineers must be able
to define the names of domain-expressions. In order to encode these knowledge roles, it must be possible to give different names (for different knowledge roles) to syntactically similar expressions. This departs from standard constructions in meta-logic (e.g. [10]) where the meta-names of object-expressions depend only on the syntactic form of the expressions.

To achieve definable names, (ML)² allows 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 knowledge roles. Typically, for any knowledge rule r, we introduce a function symbol r() in the meta-theory.

We refer to Section 9 for some examples of lift-definitions².

Representing primitive inference actions: The second aspect of an inference layer are the primitive inference actions (PIA’s). Such PIA’s map a number of input knowledge roles onto a single output knowledge role. In (ML)², inference actions are represented by meta-level theories of a restricted form. An inference action PIAk corresponds to a theory with axioms of the form³

\[
\text{LHS}_{\text{PIA}_k} \rightarrow \text{PIA}_k(t_1, \ldots, t_n, t_{n+1})
\]

The left-hand side LHS_{PIA}_k can be an arbitrary formula constructed from reflective predicates and predicates of the form input_{DKR}_k(t_i), and each t_i will be a term whose outermost function symbol represents the knowledge role DKR_i, along the lines defined above. We will postpone the definition of the input_{DKR}_k predicates to Section 5. We call the predicate PIA_k the PIA predicate. Such a PIA predicate, axiomatized by formulae of form (1), represents the primitive inference action as an (n + 1)-place relation between the n input knowledge roles and the single output knowledge role.

We refer to Section 9 for examples of primitive inference actions.

Reflection rules in (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 (ML)², we require three inference rules between meta- and object-layers, generally known as reflection rules:

\[
\frac{\phi \in \mathcal{O}}{\text{ask}^O ([\mathcal{O}], [\phi]) \text{ (axiom)}}
\]

Rule (up) states that if a formula φ is provable in the object-theory O, then the formula \text{ask}^O ([\mathcal{O}], [\phi])³³ is provable in the meta-theory \mathcal{M}, allowing inferences in

²Section 9 calls lift-definitions \#kr’s.
³or any formulation that is logically equivalent to this form
⁴where the meta-term [φ] is the name for the object-formula φ, as defined through lift-definitions.
\( O \) to affect inferences in \( M \). Conversely, rule (\textsc{down}) allows inferences in \( M \) to affect inferences in \( O \). Finally, rule (\textsc{axiom}) states that if formula \( \phi \) is an axiom of \( O \), then \( \text{ask}^\mathcal{E}(\mathcal{O},[\phi]) \) is provable in \( M \).

5 The task layer in (ML)$^2$

The purpose of the task-layer in a KA\$S\$ model is to enforce control over the inference actions 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.

In (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]. In earlier versions of (ML)$^2$, we modelled the task layer as another meta-layer, namely as the meta-theory of the inference-layer. We moved to a QDL formalism because we found first order meta-theories an insufficient device for dealing with the notions of state and sequence.

Before describing the use of QDL in (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 thus conceived as a transformation between pairs of states, transforming an initial state into the 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 \( \alpha \) and \( \beta \) are programs and \( \phi \) is a predicate, then the following are also programs:

- \( \alpha ; \beta \) (pronounced “\( \alpha \) then \( \beta \)”): do \( \alpha \) followed by \( \beta \);
- \( \alpha \cup \beta \) (pronounced “choose \( \alpha \) or \( \beta \)”): do either \( \alpha \) or \( \beta \), nondeterministically;
- \( \alpha^* \) (pronounced “repeat \( \alpha \)”): repeat \( \alpha \) a nondeterministic but finite number of times;
- \( \phi? \) (pronounced “test \( \phi \)”): proceed if \( \phi \) is true, else fail.

These elementary constructs allow the definition of various traditional programming constructs such as if-then-else, while-do, repeat-until and case statements.

The final new ingredient of QDL is a modal operator \( \langle\alpha\rangle \) for every program \( \alpha \). The compound formula \( \langle\alpha\rangle \phi \) has the following intended meaning: \( \phi \) is true in at least one terminal state of \( \alpha \). We abbreviate \( \neg\langle\alpha\rangle\neg\phi \) to \( [\alpha]\phi \) which is intended to mean: \( \phi \) is true in all terminal states of \( \alpha \).

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 \( \alpha \), and \( \phi \) 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 inference steps from the inference layer should be “executed”. QDL’s test-operator “?” allows us to turn the declarative representation of an inference action (as the \((n+1)\)-place relation \( \text{PIA}_k \) 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” inference actions, we require a representation of the state of the inference process. We use QDL variables for this purpose as follows: for each inference action \( \text{PIA}_i \), we assume a QDL variable \( V_{\text{PIA}} \) whose value will be a tuple of all input/output relations that have been computed so far for inference action \( \text{PIA}_i \). Thus, if inference action \( \text{PIA}_1 \) is a 3-place relation (two input knowledge roles, one output knowledge role), the value of \( V_{\text{PIA}} \) after calling \( \text{PIA}_1 \) three times will be \([\text{in}_{1,1}, \text{in}_{1,2}, \text{out}_1], [\text{in}_{2,1}, \text{in}_{2,2}, \text{out}_2], [\text{in}_{3,1}, \text{in}_{3,2}, \text{out}_3] \) where the \text{in}_{\text{n},\text{i}} and \text{out}_{\text{n}} are all terms from the inference layer satisfying the \( \text{PIA} \) predicate \( \text{PIA}_1 \): \( \text{PIA}_1([\text{in}_{1,1}, \text{in}_{1,2}, \text{out}_1]) \) for \( n = 1, 2, 3 \).

Furthermore, for each knowledge role \( \text{DKR}_1 \) we assume a QDL variable \( V_{\text{DKR}} \) whose value will be the tuple of all values that have been computed for knowledge role \( \text{DKR}_1 \). Thus, if \( \text{DKR}_1 \) is the output knowledge role of inference action \( \text{PIA}_1 \) above, the value of \( V_{\text{DKR}} \) after calling \( \text{PIA}_1 \) three times will be \([\text{out}_1, \text{out}_2, \text{out}_3] \).

The entire state of an inference process is now represented by the collection of all variables \( V_{\text{PIA}} \) and \( V_{\text{DKR}} \) (one variable for every primitive inference action and for every knowledge role).

**Primitive operations on PIA’s:** The above representation of the state of the inference process allows us to define the following four primitive operations on any PIA \( \text{PIA}_i ([I, O]) \):\(^6\)

- **has-solution-PIA_1([I, O])** is true if the tuple \([I, O]\) satisfies the PIA predicate \( \text{PIA}_1 \). This operation is independent of the current state of the inference process. It simply checks whether \( \text{PIA}_1 \) holds for \([I, O]\), independent of whether this tuple has been computed before.

- **old-solution-PIA_1([I, O])** is true if the tuple \([I, O]\) has previously been computed as the result of “executing” \( \text{PIA}_1 \). The predicate **old-solution-PIA_1** uses the representation of the state of the inference process as represented by the PIA variable \( V_{\text{PIA}} \), to see whether \([I, O]\) has previously been computed.

\(^6\)We write \( \Gamma \) as an abbreviation for a sequence of variables \( l_1, \ldots, l_n \). Thus, \( \text{PIA}_1([\Gamma, O]) \) is the PIA predicate for a primitive inference action with \( n \) input knowledge roles (corresponding to \( l_1, \ldots, l_n \) and an output knowledge role corresponding to \( O \)).
• more-solutions-\(\text{PIA}_i(\overline{I}, O)\) is true iff the tuple \(\overline{I}, O\) is a previously uncomputed solution to \(\text{PIA}_i\), more-solutions-\(\text{PIA}_i(\overline{I}, O)\) can be defined as

\[
\text{more-solutions-\(\text{PIA}_i(\overline{I}, O)\)} \leftrightarrow \text{has-solution-\(\text{PIA}_i(\overline{I}, O)\)} \land \neg \text{old-solution-\(\text{PIA}_i(\overline{I}, O)\)}
\]

• give-solution-\(\text{PIA}_i(\overline{I}, O)\) is true iff the tuple \(\overline{I}, O\) is a previously uncomputed solution (as with more-solutions-\(\text{PIA}_i\)), but the new solution will also be recorded in the state of the inference process. Thus, this operation corresponds to “calling” an inference action 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 \textit{predicates} of QDL, and \textit{give-solution-PIA}_i is the only operation that corresponds to a \textit{program} in QDL. Recording of the new solution in the state is performed according to the definition of the \textit{output-program output}_{\text{PIA}} that must be specified at the task layer for each \text{PIA}.

Notice that the execution of this program does not specify in any way in which order the different solutions to \(\text{PIA}_i\) will be computed. This is in accordance with the principle in KADS that inference actions are computational 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 \textit{has-solution-\(\text{PIA}_i\)}, \textit{old-solution-\(\text{PIA}_i\)}, \textit{more-solutions-\(\text{PIA}_i\)} and \textit{give-solution-\(\text{PIA}_i\)} (for each inference action \text{PIA}_i).

Using the semantics of QDL, we see that a task in \(\mathcal{M}^2\) is thus a program that maps one state of the inference process onto another state, with the state being represented by the collection of \text{PIA} variables \(V_{\text{PIA}}\), whose values contain the computed I/O tuples of all inference actions \(\text{PIA}_i\).

The \textit{input}_{\text{DKR}} \text{ predicates:} In Section 4, we used predicates of the form \textit{input}_{\text{DKR}}(t_i) in the axioms for the \text{PIA} predicates. These predicates represent the input knowledge roles \(\text{DKR}_i\) to the inference action. In \(\mathcal{M}^2\), the contents of a knowledge role can be obtained in two ways: since knowledge roles are descriptions of (the role of) domain expressions, we can retrieve the contents of knowledge roles by referring to the contents of domain theories. In this case, the \textit{input}_{\text{DKR}} predicate can be defined as

\[
\forall x : \text{input}_{\text{DKR}}(x) \leftrightarrow \text{ask}^6(O, x)
\]

where \(O\) is (the name of) the object-theory mentioned in the left-hand side of the rewrite rules in the lift-operator for knowledge role \(\text{DKR}_i\).\footnote{Formula (2) might suggest that \text{ask}^6 is the only predicate used in formulating KADS models in \(\mathcal{M}^2\). However, the reader should remember that other reflective predicates, notably \text{ask}^4, can occur in the bodies of \text{PIA} predicates, as specified in Section 4.}

Alternatively, we can retrieve the contents of knowledge roles from the \(V_{\text{DKR}}\) variables used to store the state of the inference process:

\[
\forall x : \text{input}_{\text{DKR}}(x) \leftrightarrow \exists y : V_{\text{DKR}} = [x]y
\]
if we are interested in the most recently computed value, or
\[ \forall x: \text{input}_{D\text{KR}i}(x) \rightarrow x \in V_{D\text{KR}i} \]
if we are interested in any previously computed value\(^8\), or
\[ \forall x: \text{input}_{D\text{KR}i}(x) \rightarrow x = V_{D\text{KR}i} \]
if we want all previously computed values\(^9\).

Thus, our formalism allows for any of the multiple uses that are often made of the contents of knowledge roles in KADS models, but forces the user to make clear in which way each knowledge role is used.

An example task layer can be found in Section 10.

6 Summary and conclusions

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. The situation can be summarised as follows:

\[ \begin{array}{ccc}
\text{DOMAIN} & \longrightarrow & \text{INFECTION} \\
\uparrow & & \uparrow \\
lifts & \subset & \text{TASK} \\
\text{meta} & \subset & \text{STRATEGY} \\
\end{array} \]

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

---

\(^8\)We abuse notation somewhat: the \(V_{D\text{KR}i}\) are represented as lists, but we often use set-like notation (e.g. \(\in\)) when this is convenient.

\(^9\)Notice that these different definitions of the \(\text{input}_{D\text{KR}i}\) predicate result in different possible types for the various \(\text{input}_{D\text{KR}i}\) predicates.
Part II: Conceptual model of the scheduling task

7 Introduction

As discussed above, a KADS conceptual model consists of four layers: the domain layer, the inference layer, the task layer and the strategy layer. We have constructed a conceptual model consisting of the first three layers only, as there is no strategic reasoning in the problem at hand. Although various ideas come to mind as to how the assignment problem can be solved in other, more efficient ways than the prescribed solution, we stuck to the given solution as close as possible. The resulting inference structure is shown in Figure 1.

![Inference Structure](image)

Figure 1: The inference structure for the assignment task.

The task divides naturally into three steps: a first step that proposes a refinement...
to the current schedule, a test step that evaluates the resulting schedule, and a revise step that modifies the current schedule if necessary. These 3 steps are indicated by the dashed boxes in Figure 1. Each of these steps is further divided into subtasks which each correspond to a step in the example problem solving trace. The six subtasks listed in the description of the complex reasoning task [9] are modelled by six inference steps:

requirements generation This task embodies interaction with the user and is therefore considered a transfer task. Transfer tasks are not modelled at the inference layer, hence there is no corresponding inference step.

assumption generation This subtask is modelled at the inference layer by the generate assignments inference step. Its input is the current object description, a set of consumers and a set of providers, and its output is a set of assignments. The consumers and providers in this domain are the activities and time-slots respectively. We have generalised these entities to make the inference layer reusable for other assignment tasks.

assumption selection The inference step select assignment models this subtask. Input to the inference step are the set of assignments, and the output is the selected assignment. Selection is done on the basis of selection heuristics, which in KADS is called a static knowledge role or support role. This means that the selection heuristics are not changed during inferencing, but that they are used to guide the inferencing. The static knowledge roles are formally represented in the same way as dynamic knowledge roles.

design object evaluation This subtask is accomplished by two inference steps: add assignment, that actually adds the proposed assignment as an assignment to the object description, and the inference step evaluate object, which checks whether a conclusion can be drawn from the resulting object description about the requirements. The output role conclusion does not correspond to any domain knowledge, as it is used solely to decide which step is to be taken next.

design process evaluation If conclusion indicates that a requirement has been violated then the design process is evaluated. This is done by the inference step evaluate process. The inference step takes the object description as input, and determines the culprit for the violation. In this example, the culprit is the last assignment that was added to the object description. The assignment and the context in which the assignment was a culprit are both modelled by the knowledge role culprits and contexts.

assumption revision The inference step revise assignment takes the culprit and context and removes the culprit and its dependencies from the object description. Since the heuristic is used that the culprit is the most recent assignment, there are no assignments dependent on this assignment. This heuristic is not expressible in terms of domain knowledge alone, since it refers to the state of the problem solving process, and hence it is not possible to model this heuristic as a knowledge role.
8 Domain knowledge

In this section we describe the knowledge at the domain level. The structure of the formal domain knowledge is depicted in Figure 2. The boxes denote theories and the arrows denote import relations, e.g. theory `worldKnowledge` imports theory `language`.

![Figure 2: Structure of the domain knowledge](image)

Now we have to describe the contents of the theories. First we define a general theory that will be imported by all other domain theories and that defines a common language.

```plaintext
theory language
  signature
    sorts activity, period ;
  predicates
    @ : activity × period ;
    = : universal × universal ;
    < : period × period ;
end-theory
```

Note that "<" is defined on `period×period`, not on activities. The ordering on activities is induced by their assignment to periods.

There are some general facts (so-called world knowledge) about time. For instance there is transivity, anti-symmetry and non-reflexivity of the "<" predicate on time periods. Determinacy is defined for the assigment of activities on time periods.
theory worldKnowledge
  import language ;
  axioms
  \( \forall T_1 : \text{period } \neg (T_1 < T_1) ; \)
  \( \forall T_1, T_2 : \text{period } T_1 < T_2 \rightarrow \neg (T_2 < T_1) ; \)
  \( \forall T_1, T_2, T_3 : \text{period } (T_1 < T_2) \land (T_2 < T_3) \rightarrow T_1 < T_3 ; \)
  \( \forall T_1, T_2 : \text{period } \forall A : \text{activity } (A \@ T_1) \land (A \@ T_2) \rightarrow T_1 = T_2 ; \)
end-theory

The next theory contains information that is specific for a special case, in KADS this is referred to as **case-data**. The given activities and time periods are declared as constants, and the axioms express the particular ordering for time periods for this case.

theory caseData
  import worldKnowledge ;
  signature
    constants
      a_1, a_2, a_3, a_4 : activity ;
      t_2, t_1, t_3 : period ;
  axioms
    t_1 < t_2 ;
    t_2 < t_3 ;
end-theory

Finally we can define a theory which states the particular requirements for this case. Note that all other theories are imported here (by transitivity of the import relation) to provide the necessary vocabulary. The requirements (e.g. those from requirements set one) are expressed as axioms of this theory.

theory domainRequirements
  import caseData ;
  axioms
    \( \forall T_1, T_2 : \text{period } (a_3 \@ T_1) \land (a_2 \@ T_2) \rightarrow T_1 < T_2 ; \)
    \( \forall T_1, T_2 : \text{period } (a_3 \@ T_1) \land (a_4 \@ T_2) \rightarrow T_1 < T_2 ; \)
    \( \forall T_1, T_2 : \text{period } (a_2 \@ T_1) \land (a_4 \@ T_2) \rightarrow (T_1 = T_2) ; \)
end-theory

There is one theory left to define and that is the theory that will eventually contain the solution of the problem after the reasoning has been completed. Although no axioms are defined, the language to express the solution must be defined. This is accomplished by importing the theory that defines the language.

theory solution
  import language, caseData ;
end-theory

9 Inference layer

The inference layer is formalised as a set of theories and lift-operators, with the import- and use-structure between these theories and lift-operators closely following the topology of the inference structure from Figure 1. There are two deviations
between the import structure and the original inference structure of Figure 1: inferenceLanguage is introduced to define the globally used language elements, and is thus imported everywhere else. Similarly, the knowledge role objectDescription introduces some globally used language constructions, and is thus imported in more places than required by the inference layer topology. Note that this construction is enforced by the (ML) requirement that language elements are only declared once.

9.1 Propose

The first step of the three-step propose-test-revise loop at the inference layer proposes a refinement to be made to the current design state. The design state is represented as the objectDescription, and a refinement is proposed by first generating all possible refinements, and then selecting and adding one of these.

As there are several modules that use the same language elements, we need one module that defines this language, which is then used by the other modules. (Note that this module has the same role as language in the domain layer.) We define this module first. It only introduces two sorts that embody the generalization of activities and periods, i.e. consumer and provider.

```
lift-definition inferenceLanguage
  signature
    sorts consumer, provider ;
  end-lift-definition
```

Next we define the DRIs that are connected to the first PIA generate. This PIA has three roles as input and one as output. From the consumers, providers and the current object description it generates a set of possible assignments. The DRIs consumers and providers define (inference layer) names for activities and periods at the domain layer.

```
lift-definition consumers
  from caseData, solution ;
  use inferenceLanguage ;
  lift-variables A : constant ;
  signature
    constants [A : activity] : consumer ;
    predicates inputConsumers : consumer ;
    mapping lift(A : activity) => [A] ;
  end-lift-definition
```

Note that the naming information is encoded in the sort of the resulting name. We could as well have encoded this information more visibly by introducing a function symbol. Note also that DRIs already declare an input predicate inputOKR, which will be used in PIA’s to obtain the contents of this DRI. The definition of this predicate will be given at the task layer (see also Section 5).

---

10This lift-definition shows an unfortunate overloading of notation in (ML): the type consumer is the sort of the constant [A]. The annotation constant in the declaration of the lift-variable A indicates that it can only match with constant symbols of domain language, and the annotation activity in the lift-rule occurrence of the variable A restricts the matches of A further to constants of domain sort activity.
The DKR providers is the dual of consumers:

```plaintext
lift-definition providers
  from caseData, solution;
  use inferenceLanguage;
  lift-variables T : constant;
  signature
    constants [T : period] : provider;
    predicates inputProviders : provider;
    mapping lift(T : period) → [T];
end-lift-definition
```

One thing that is central to the whole reasoning process is the set of assignments of consumers to providers that is being built. The term used in the example to indicate this structure is “(partial) object description”. The adjective “partial” indicates that during the reasoning process the description refers only to a part of the final object. This description changes dynamically and is modelled at the inference level during the reasoning process. At the end of this process, the solution needs to be expressed in domain terms. For this purpose, we need lift rules that will be used downward to translate the inference level description in domain specific terms. These lift-rules are given in the DKR objectDescription below. The first rule defines how assignments on the inference level will be reflected at the domain in domain specific terms. The second rule is needed because, at the inference level, we deal with lists of assignments. This means that we need a (recursive) lift rule to lift such lists. Note that we specify how to lift a list without specifying how to lift its elements. At the domain layer, a list is reflected as a conjunction of (the translations of) the list elements.

```plaintext
lift-definition objectDescription
  from solution;
  use providers, consumers;
  lift-variables
    T, A : constant;
    Others : sentence;
  signature
    sorts assigned;
    functions assignment : consumer × provider → assigned;
    predicates inputObjectDescription : assigned set;
  mapping
    lift(true) ← [];
    lift(A : activity@T : period) ← assignment(lift(A), lift(T));
    lift(A : activity@T : period ∧ Others) ←
      [lift(A : activity@T : period) | lift(Others)];
end-lift-definition
```

Because all language elements have now been defined, we are in a position to define our first PLA.

---

11 Except the lift-definition culpritPlusContext, which will be defined below.
12 From here on we delete explicit quantification of axioms. All variables are universally quantified at the front of the formula unless indicated otherwise.
theory generate
  use consumers, providers.objectDescription, culpritsPlusContext, assignments;
signature
  predicates
    already-assigned : consumer × assigned set;
    known-as-violation : assigned × culpritContext set × assigned set;
    PIA-generate : consumer set × provider set × assigned set ×
      culpritContext set × assigned set;
axioms
  PIA-generate(Consumers, Providers, ObjDescr, Culprits, Assignments) ←
    Consumers = \{ C | inputConsumers(C) \} \land (1)
    Providers = \{ P | inputProviders(P) \} \land (2)
    inputObjectDescription(ObjDescr) \land (3)
    inputculpritsPlusContext(Culprits) \land (4)
    Assignments = \{ assignment(C, P) | C \in Consumers \land P \in Providers \land
    ¬ already-assigned(C, ObjDescr) \land (5)
    ¬ known-as-violation(assignment(C, P), Culprits, ObjDescr) \}; (7)

  already-assigned(Consumer, ObjDescr) ←
    \exists Provider : provider assignment(Consumer, Provider) \in ObjDescr;

  known-as-violation(Assignment, Culprits, ObjDescr) ←
    \exists ObjDescr_1 : assigned set
      ObjDescr_1 \subseteq (ObjDescr \cup \{ Assignment \}) \land
      culprit(ObjDescr_1, Assignment) \in Culprits;
end-theory

This axiom states that, based on the given consumers (line 1), producers (line 2),
the current object-description (line 3) and the known culprits (line 4), we construct
all pairs of consumers and producers (line 5) that are not yet assigned (line 6) and
that are not yet known as violations (line 7).

Notice that culprits are not single assignments (as suggested in the example
text), but are instead pairs of a partial object description and an assignment,
because an assignment is only a culprit in the context of a given partial object
description.

The above PIA encodes the heuristic as stated in the example that we generate
all possible assignments for all not yet allocated activities except those that are
already known to be a violation. Notice that this heuristic is not represented as a
separate knowledge role, but instead encoded in the axiom of the PIA. This is done
because it depends on the notions of “not already assigned” and “not known to be
a culprit”. These notions are inference-layer concepts, and do not correspond to
any declaratively true domain knowledge. As a result, the heuristic that relies on
these notions cannot be encoded as a description of domain knowledge (i.e., a DKR),
but instead as knowledge that only lives at the inference layer (i.e., as part of the
axiom set of a PIA).

The following defines the DKR (possible) assignments. This DKR does not appar-
tently introduce any new language elements, because it will refer to sets of assign-
ments, while the assignments have already been defined in the DKR objectDescription
and the set constructor is built into \( \mathcal{M}\).

\[
\text{lift-definition assignments}
\]
\[
\begin{align*}
\text{use } & \text{objectDescription} ; \\
\text{signature} & \\
\text{predicates} & \text{input assignments : assigned set} ; \\
\end{align*}
\]

end-lift-definition

The next \texttt{DKR} will represent the heuristics used to select the next assignment from all possible assignments (those generated by the \texttt{PIA generate}).

\[
\text{lift-definition selectionHeuristics}
\]
\[
\begin{align*}
\text{from } & \text{caseData} ; \\
\text{use } & \text{objectDescription} ; \\
\text{lift-variables} & T_2, T_1 : \text{constant} ; \\
\text{signature} & \\
\text{sorts} & \text{preferred} ; \\
\text{constants} & [T_1 : \text{period}], [T_2 : \text{period}] ; \text{provider} ; \\
\text{functions} & \text{preference : provider} \times \text{provider} \rightarrow \text{preferred} ; \\
\text{predicates} & \text{input selectionHeuristics : assigned set} ; \\
\text{mapping} & \text{lift}(T_1 : \text{period} < T_2 : \text{period}) \rightarrow \text{preference(lift}(T_1), \text{lift}(T_2)) ; \\
\end{align*}
\]

end-lift-definition

This lift-rule encodes the preference of earlier time slots over later ones. This domain specific preference criterion is abstracted into a domain independent representation. Note that we could make the preference schema more general by including the activities as arguments: \( \text{preference}(A_1, T_1, A_2, T_2) \). This would cater for selection heuristics which also prefer certain activities over others on other grounds than the time at which they are scheduled.

The next \texttt{PIA} will use the preference criterion to select an assignment from the set of possible assignments. We pick an element \( \text{assignment}(A_1, T_1) \) from the set of possible assignments such that all other elements from this set are less preferred than the selected element.

\[
\text{theory selectAssignment}
\]
\[
\begin{align*}
\text{use } & \text{assignments, selectionHeuristics} \ ; \text{assignment} ; \\
\text{signature} & \\
\text{predicates} & \text{other-preferred : assigned set} \times \text{assigned} ; \\
\text{PIA-selectAssignment : assigned set} \times \text{assigned} ; \\
\text{axioms} & \\
\text{PIA-selectAssignment}(\text{AssS, Ass}) \leftarrow \\
\text{input assignments(\text{AssS})} \land \\
\text{assignment}(A_1, T_1) \in \text{AssS} \land \\
\forall A_2 : \text{consumer} \forall T_2 : \text{provider} \\
\text{assignment}(A_2, T_2) \in \text{AssS} \rightarrow \text{ask}^+(\text{caseData, preference}(T_2, T_1)) ; \\
\end{align*}
\]

end-theory

Notice that we use inference at the domain layer (through \( \text{ask}^+ \)) to deduce the preference relation which is based on ordering of time slots. This enables the use of
general knowledge about the domain (such as the transitivity of the time-ordering),
without specifying any further control over this inference process within the PIA.

The following DKR and PIA are rather trivial: the DKR assignment defines no
new language elements except its own input predicate, and the PIA add only states
how to add a given assignment to the current objectDescription.

lift-definition assignment
  use objectDescription ;
  signature
    predicates input assignment : assigned ;
end-lift-definition

theory add
  use assignment ; objectDescription ;
  signature
    predicates, PIA-add : assigned × assigned set × assigned set ;
  axioms
    PIA-add(Ass, ObjDescr, ObjDescr₁) ←
      input assignment(Ass) ∧
      input objectDescription(ObjDescr) ∧
      ObjDescr₁ = [Ass | ObjDescr] ;
end-theory

9.2 Test

The purpose of the second step in the propose-test-revise cycle is twofold: firstly
to test whether the proposed refinement of the current objectDescription violates
any of the design constraints, and secondly to test if the propose-test-revise cycle
should be terminated (because the design process has finished).

The main DKR involved here encodes these two pieces of domain knowledge (the
violation and termination criteria) in a domain independent way:

lift-definition requirements
  from domainRequirements ;
  use objectDescription, conclusion ;
  lift-variables
    A, S : variable ;
  signature
    functions
      violationObject, finishedObject : assigned set → controlConstant ;
    mapping
      lift(S → ⊥) ← violationObject(lift(S)) ;
      lift(S → ∀ A : activity ∃ T : period (A ⊚ T) ← finishedObject(lift(S)) ;
end-lift-definition

The domain specific violation criterion in this example task is inconsistency of
the current objectDescription with the domain requirements (first rule above), and
the termination criterion is the assignment of all activities to a given time slot
(second rule above).
Note that the mapping rules have a non-ground term at the right hand side. We use DKR objectDescription that contains the mapping rules to handle this non-ground term, i.e., lift($S$)$^{13}$.

The next DKR will be used to encode the result of the test operation: we introduce two constants, one for indicating violation, and one for indicating termination. This DKR will be used at the task layer to control the execution of the propose-test-revise loop: if a violation is detected, the revise step is activated; if termination is detected, the reasoning process halts, otherwise the next propose step is called.

```
lift-definition conclusion
  use objectDescription;
signature
    sorts controlConstant;
    constants violation, finished : controlConstant;
end-lift-definition
```

The only PIA of the test part of the inference structure is given below:

```
theory evaluateObject
  use requirements, objectDescription, conclusion;
signature
  predicates PIA-evaluateObject : assigned set \times controlConstant;
axioms
  PIA-evaluateObject(ObjDescr, violation) \leftarrow
    input(objectDescription)(ObjDescr) \land
    ask$^b$(domain Requirements, violationObject(ObjDescr));

  PIA-evaluateObject(ObjDescr, finished) \leftarrow
    input(objectDescription)(ObjDescr) \land
    \neg ask$^b$(domain Requirements, violationObject(ObjDescr)) \land
    ask$^b$(domain Requirements, finishedObject(ObjDescr));
end-theory
```

These two axioms use the language elements introduced above and define how to detect termination and violation. Again (as in the PIA selectAssignments), this is done by exploiting inference at the domain layer through the ask$^b$ predicate. This uses the specification given in DKR requirements that violation amounts to inconsistency and termination to allocation of all activities. Again, no further internal control is enforced on these two inference processes that take place at the domain layer.

9.3 Revise

The final step of the propose-test-revise cycle decides which of the possible assignments must be withdrawn in case of a detected violation. This amounts to the detection of a culprit, the representation of which is given in the following DKR:

$^{13}$Because the type sentence of the lift-variables $A$ and $T$ are taken from a predefined set of possible types for such variables (just as the type constant in previous lift-definitions), such types do not need not be declared before they can be used.
lift-definition culpitsPlusContext
  use objectDescription;
  signature
  sorts culpirtContext;
  functions culpirt : assigned set \times assigned \rightarrow culpirtContext;
  predicates input culpitsPlusContext : culpirtContext set;
end-lift-definition.

Notice that culpits are not single assignments (as suggested in the example
text), but are instead pairs of a partial object description (i.e. a set of assignments)
and an assignment, because an assignment can only be a culpirt in the context of a
given partial object description.

The following pIA represents the extremely simple heuristic that the most recent
assignment is the proper candidate for rejection (as specified in the example text).
Again, this heuristic is not modelled as a knowledge role, since it is dependent on
aspects of the reasoning process itself (the most recent assignment), and thus cannot
be expressed at the domain layer (and subsequently lifted as a DKR at the inference
layer), but should instead be expressed at the inference layer only.

theory evaluateProcess
  use objectDescription ; culpitsPlusContext;
  signature
  predicates PIA-evaluateProcess : assigned set \times culpirtContext ;
  axioms
  PIA-evaluateProcess(ObjDescr. Culprit) \leftarrow
  \exists MostRecentAssignment : assigned
  \input_{objectDescription} (ObjDescr) \land
  ObjDescr = [MostRecentAssignment \land]
  Culprit = culpirt(ObjDescr. MostRecentAssignment);
end-theory.

The final trivial pIA states how to remove the culpirt, generated by the above
PIA evaluateProcess according to the revision heuristics.

theory revise
  use objectDescription, culpitsPlusContext;
  signature
  predicates PIA-revis : assigned set \times culpirtContext \times assigned set ;
  axioms
  PIA-revis(ObjDescr, Culprit, ObjDescr1) \leftarrow
  \exists Context : assigned set
  \exists C : assigned
  \input_{objectDescription} (ObjDescr) \land
  \input_{culpitsPlusContext}([Culprit \land]) \land
  Culprit = culpirt(Context, C) \land
  ObjDescr = (ObjDescr1 \cup \{ C \});
end-theory.
10 Task Knowledge

This section will present the task knowledge. The task layer first of all specifies the decomposition of the top level task into smaller tasks, the leaves of this decomposition being references to the inference actions at the inference layer. The task layer also specifies the control over these subtasks. Notice that here we use Quantified Dynamic Logic (QDL). So we have two principal ingredients: predicates and programs.

As mentioned before, the general problem has been decomposed into three subtasks:

1. propose an assignment
2. test whether the assignment fulfills the requirements
3. revise the assignments if needed

As the task layer specifies the dynamics of the inference structure, it is at this layer that we specify what knowledge the knowledge roles point. This is done by defining the input and output predicates for each knowledge role in the inference structure. With each knowledge role a variable is associated, which contains the knowledge the role points to. At the task layer, these variables can be inspected to establish the input of a PIA, or they can be altered, to store the results of a PIA. Each PIA also has a task variable associated with it, containing the input/output tuples that have been computed. These variables are used to generate new solutions, i.e. solutions that have not been computed before.

Remember that the task layer is expressed in dynamic logic, which adds to standard logic the notion of variables (to encode state) and program (to encode action).

10.1 Definitions of programs to update the output knowledge roles

The output of each inference action is prepended to the task variable associated with the output knowledge role. Hence for each knowledge role KR we have an output program\(^{14}\):

\[
\text{output}_{DKR}(X) \equiv (V_{DKR} := X)
\]

10.2 Definition of predicates to access input knowledge roles

The input predicates refer either to the variables associated with the input role, when the input role is an intermediate one, i.e. when the knowledge it refers to is dynamically derived, or it refers to the domain layer, when the role points to static domain knowledge.

\(^{14}\)\(\&\) is the prepend operation on lists.
The consumers and providers DKNs refer to axioms in the caseData domain module:

\[
\forall X : \text{consumer}(\text{input}_\text{consumers}(X) \leftrightarrow \text{ask}^\bowtie (\text{caseData}, X))
\]

\[
\forall X : \text{provider}(\text{input}_\text{providers}(X) \leftrightarrow \text{ask}^\bowtie (\text{caseData}, X))
\]

The intermediate roles objectDescription, assignments and assumption refer to their knowledge role variables:

\[
\forall X : \text{assigned set} (\text{input}_\text{objectDescription}(X) \leftrightarrow V_{\text{ObjectDescription}} = [X|])
\]

\[
\forall X : \text{assigned set} (\text{input}_\text{assignments}(X) \leftrightarrow V_{\text{assignments}} = [X|])
\]

\[
\forall X : \text{assigned} (\text{input}_\text{assignment}(X) \leftrightarrow V_{\text{assignment}} = [X|])
\]

For the knowledge role culprits we have a variation of the above:

\[
\forall X : \text{culpritContext set} (\text{input}_\text{culpritsPlusContext}(X) \leftrightarrow X = V_{\text{culprits}})
\]

since all of the (past) culprits are taken into account, and not only the most recent one.

10.3 Definition of the task expression

The overall task expression specifies the control over the execution of the inference actions:

\[
\begin{align*}
\text{initialise} & ; \\
\text{WHILE} \ (\neg \text{test(ObjDescr, finished)}) \ & \text{DO} \ \text{propose(ObjDescr, NewObjDescr)} ; \\
& \quad \text{IF test(NewObjDescr, violation)} \\
& \quad \quad \text{THEN revise-task(NewObjDescr, RevisedObjDescr)} \quad \text{(T)} \\
& \quad \quad \text{FI} \\
& \quad \text{OD} ; \\
& \quad \text{tell(solution, ObjDescr)} ?;
\end{align*}
\]

The object description has to be an empty description when we start execution, since generateAssignment has to decide which assignments have already been generated (namely none so far). Hence:

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

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

\[
\text{propose(ObjDescr, NewObjDescr)} \equiv \\
\quad \text{give-solution-generate(Cons, Prov, ObjDescr, Culprits, AssS)} ; \\
\quad \text{give-solution-selectAssignment(AssS, Ass)} ; \\
\quad \text{give-solution-add(Ass, ObjDescr, NewObjDescr)}
\]
Next the proposed assignments are tested to see if any requirement is violated:\(^5\):

\[
\text{test}(	ext{ObjDescr, Value}) \rightarrow \text{has-solution-evaluateObject}(	ext{ObjDescr, Value}); \quad (5)
\]

Now, if any requirement has been violated, the assignment set is revised:

\[
\text{revise-task}(	ext{ObjDescr, Revised ObjDescr}) \equiv \\
\quad \text{give-solution-evaluateProcess}(	ext{ObjDescr, Culprit}); \\
\quad \text{give-solution-revise}(	ext{ObjDescr, Culprit, RevisedObjDescr})
\]

Note that the \textit{give-solution} and \textit{has-solution} predicates are mechanically derived for each inference action. Finally, when the \textit{while} loop finishes, we need to translate the solution, which is stated in inference layer terms, back to the domain layer. This is accomplished by the reflective predicate \textit{tell}. This completes the description of the task layer, and therefore of the formal model of the assignment task.

11 Conclusions

11.1 Evaluation

When looking at our description of the simple design task, the main negative feature that strikes us is the amount of code needed to express fairly simple concepts. A full-length description of the \texttt{ML}\textsuperscript{2} code for the example task runs to 400 lines, of which only 100 contain interesting information (axioms, lift-rules). This redundancy is partly caused by a rather richly sugared syntax, and partly because the strong typing of \texttt{ML}\textsuperscript{2} requires many declarations, which are often repetitive. We are convinced that good support from dedicated editors can relieve much of this burden: much syntactic sugar can be provided through templates, and much of the type declarations can be inferred, and need not be explicitly stated each time by the user.

We have already developed TheME \cite{2}, a dedicated editor for \texttt{ML}\textsuperscript{2} which allows easy navigation through the \texttt{ML}\textsuperscript{2} code for a conceptual model, and which also “knows” enough about KADS models to actively support the user in the modelling process.

A final feature that is striking about the language is that it allows partial specifications because the \texttt{qDL} formalism of the task layer allows for non-deterministic choice among program statements. The problem solving trace which follows in the appendix amply illustrates this feature of the language.

11.2 Comparison with other languages

When comparing \texttt{ML}\textsuperscript{2} with other contributions to this volume \cite{11}, a first point to be noticed is that \texttt{ML}\textsuperscript{2} takes a rather extreme position on the scale of operational and/or formal languages. \texttt{ML}\textsuperscript{2}’s explicitly stated aim is to specify KADS models

\(^5\text{We use the } \equiv \text{ symbol for defining programs, and the } \rightarrow \text{ symbol for defining predicates.}\)
using a collection of well understood mathematical constructions, without particularly worrying about executability of these constructs. Thus, the aims of (ML$^2$) are in at this point in contrast with those of languages like AIDE and MILORD.

A second point to notice is the use of dynamic logic to express control. Almost all languages described in this volume contain some constructs for specifying the dynamic behaviour of systems, (ML$^2$) is the only language using dynamic logic for this purpose (although other languages such as KARL have recently been moving in the same direction [7]). The advantage of using dynamic logic is that it gives an overall declarative semantics for the control language, in contrast with some of the other languages in this volume which use either a fully procedural language, or use a language which is locally declarative, but have a procedural interpretation of how these declarative components are strung together to obtain actual behaviour.

Finally, (ML$^2$) differs from many of the languages in this volume because it is based on an underlying prescriptive model, namely KADS. It shares this feature with KARL [7], and differs from languages like DESIRE [8] and KBSSF [12], which, although they can be used to specify KADS models are not specifically geared to do so.

11.3 Future Work

Currently, the (ML$^2$) framework is logically sound and it is sufficiently expressive to specify KADS models of expertise. In fact, the expressivity does not restrict the model builder to KADS models alone. Therefore we are working on a so-called Kadsified version of (ML$^2$). This kadsification has three goals:

1. enhance the readability of (ML$^2$) specifications for non-mathematically oriented people, e.g. knowledge engineers;

2. limit the expressivity of (ML$^2$) constructs to KADS models of expertise and

3. distinguish (ML$^2$) constructs that have different semantics in KADS models (but of course the same logical semantics).

These goals are realized through introducing extra syntax in some places (to allow finer distinctions between elements that have the same logical semantics but that play different roles in KADS models), and by restricting the syntax in other places (to exclude non-KADS-like constructions).

An ongoing activity is the development of support tools for making formal models. Currently we have a dedicated editing environment ([2]), a parser and type-checker, and an interpreter for (ML$^2$) models (called $\mathcal{S(ML^2)}$). The latter enables the assessment of properties of the model by simulation. It is not intended as a prototyping or implementation environment.

Recent ideas in the KADS-II project have lead to a refinement of ideas on how to separate more clearly application knowledge from problem-solving knowledge [14]. These ideas are currently being formalized.

Another interesting application of formal (ML$^2$) specifications stems from work in the REFLECT project. Here a strategic component for KBS is realized as a
reflective component, i.e. as a separate system with the KBS as its domain. Instead of working directly on the KBS, the strategic component worked on an $\text{ML}^2$ specification of the KBS. For details see [6].

Finally we are very much interested in the applicability of formal knowledge models, especially by users outside the community that developed $\text{ML}^2$. Recently a comprehensive set of guidelines for building formal knowledge models from informal ones has been produced. We expect that these guidelines will make formal modelling more accessible. Also these guidelines show that much of the formal description can be generated by knowledgeable tools, leaving only those parts that really incorporate new or extended knowledge.

References


**Appendix: trace for example No. 1**

\( \text{ML}^2 \) is a specification language for knowledge-based systems. As expected from any specification language, \( \text{ML}^2 \) specifies the desired behaviour of a system without determining how such behaviour should be achieved. \( \text{ML}^2 \) does in fact not necessarily determine a unique behaviour. This allows partial behaviour specifications, as well as specifications which leave room for implementation decisions to be postponed, rather than enforcing arbitrary choices already in the specification itself.

In particular, it is possible to specify non-deterministic behaviour in \( \text{ML}^2 \) through the use of the dynamic logic programs at the task layer. The non-determinism in \( \text{ML}^2 \) arises from two sources: the non-determinism of some of the QDL operators, and from the non-determinism of the order in which the knowledge-source predicates compute their solutions. The (declarative) semantics of a non-deterministic QDL program is the set of all its terminal states.

As it happens, the specification given above is such a non-deterministic program (ie. it has more than one terminal state). Some of these terminal states contain solutions to the problem at hand, while others do not. More formally, if we use \( T \) to refer to the task expression in Section 10.3, then \( [T] \) denotes the set of all
terminal states of T. Of these possible states, those containing a solution can be characterised by

\[ T \exists X : \text{test}(X, \text{finished}) \]  

(i.e., those terminal states satisfying the termination condition specified in the PIA \text{evaluateObject}). The value of X is then regarded as the solution. A complete and sound implementation of this specification must ensure that all these terminal states (and no others) are reachable in the implementation.

For presentation purposes, in the trace below (which is a hand-simulated trace), we have assumed a Prolog-style “depth-first” choice at each choice-point, and this happens to be a path that does not lead to a solution of the given example (i.e., a terminal state of program T in which (6) does not hold). Other choices at the non-deterministic points in the program do lead to solutions of the example problem (i.e., (6) is satisfiable).

Execution of T could give the following trace. We will trace the first test step in detail and later we will give larger steps through the execution.

First of all, the subtask initialise; leads to

\[ V_{\text{ObjDescr}} := [\_] \]

by definition 4. Subsequently, we execute the while loop of expression T:

\[ \rightarrow \text{test}(\text{ObjDescr}, \text{finished}) \]

\[ \rightarrow \sim \text{has_solution} \rightarrow \text{evaluateObject}(\text{ObjDescr}, \text{finished}) \]

By definition 5

\[ \rightarrow \sim \text{evaluateObject}(\text{ObjDescr}, \text{finished}) \]

By the definition of has_solution PIA in section 5

\[ \rightarrow \sim (\text{inputObjectDescription}(\text{ObjDescr}) \land \sim \text{ask}^\ast (\text{DomainRequirement}, \text{violationObject}(\text{ObjDescr})) \land \text{ask}^\ast (\text{DomainRequirement}, \text{finishedObject}(\text{ObjDescr})) \) \]

By the second axiom of PIA \text{evaluateObject}

\[ \rightarrow \sim ([\_] = [\text{ObjDescr}]) \land \]

By definition 3

\[ \rightarrow \sim (\text{DomainRequirement} \vdash (\text{TRUE} \rightarrow \bot)) \land \]

By dkr-definitions requirements and objectDescription and (up)-rule for ask^\ast .

\[ (\text{DomainRequirement} \vdash (\text{TRUE} \rightarrow \forall A : \text{activity} \exists T : \text{timeA} @ T)) \]

\)

By the same

\[ \rightarrow \sim (\text{ObjDescr} = [\_] \land \text{TRUE} \land \bot) \]

By unification and tautology

\[ \rightarrow \text{TRUE} \]

by tautology
Hence the first test succeeds, and the body of the while–loop is executed:

\[
\text{propose}([], \text{NewObjDescr}),
\]
giving:

\[
\begin{align*}
\text{give}_\text{solution}_\text{generateAssignments}(\text{Cons}, \text{Prov}, [], \text{Culprits}, \text{AssS}) ; \\
\text{give}_\text{solution}_\text{selectAssignment}(\text{AssS}, \text{Ass}) ; \\
\text{give}_\text{solution}_\text{add}(\text{Ass}, [], \text{NewObjDescr})
\end{align*}
\]
(by the definition of propose in section 10.3. By the axiom for the PIA generateAssignments), the first statement gives:

\[
\begin{align*}
\text{Cons} & = \{ [a_1], [a_2], [a_3], [a_4] \} \\
\text{Prov} & = \{ [t_1], [t_2], [t_3] \} \\
\text{Culprits} & = [] \\
\text{AssS} & = \\
& \{ \text{assignment}([a_1], [t_1]), \text{assignment}([a_2], [t_2]), \text{assignment}([a_3], [t_3]) \\
& \quad \text{assignment}([a_2], [t_1]), \text{assignment}([a_2], [t_2]), \text{assignment}([a_2], [t_3]) \\
& \quad \text{assignment}([a_3], [t_1]), \text{assignment}([a_3], [t_2]), \text{assignment}([a_3], [t_3]) \\
& \quad \text{assignment}([a_4], [t_1]), \text{assignment}([a_4], [t_2]), \text{assignment}([a_4], [t_3]) \}
\end{align*}
\]
After this, the second statement from (7) becomes:

\[
\text{give}_\text{solution}_\text{selectAssignment}(\{ \ldots \text{AssS above} \ldots \}, \text{Ass}) ;
\]
which, by the axiom for the PIA selectAssignment, gives:

\[
\text{Ass} = \text{assignment}([a_1], [t_1])
\]
Note that we could have selected any activity that is scheduled on t1, because of the non-determinacy of the ε in the axiom for selectAssignment.

The third statement from (7) now becomes

\[
\text{give}_\text{solution}_\text{add}(\text{assignment}([a_1], [t_1]), [], \text{NewObjDescr})
\]
and gives (by the axiom for the PIA add):

\[
\text{NewObjDescr} = \text{assignment}([a_1], [t_1])
\]
After each of the three give_solution_PIA programs from (7) above, the corresponding V_{PIA} variable is updated by prepending the tuple of all arguments to its current value, and the V_{PIR} variables for the output roles are updated by prepending the computed output value, e.g.:

\[
\text{V}_{\text{ObjDescr}} = [[\text{assignment}([a_1], [t_1])], []]
\]
In the subsequent execution of the main task expression ($T$), the statement
\[
\text{test(assignment([a_1], [t_1]), violation)}
\]
fails, leading to the next iteration in the while-loop:
\[
\text{test(assignment([a_1], [t_1]), finished)}
\]
fails by execution along the lines outlined above, so the loop continues:
\[
\text{propose(assignment([a_1], [t_1]), NewObjDescr)}
\]
generates:
\[
\{ \quad \text{assignment([a_2], [t_1]), assignment([a_2], [t_2]), assignment([a_2], [t_3])} \\
\text{assignment([a_3], [t_1]), assignment([a_3], [t_2]), assignment([a_3], [t_3])} \\
\text{assignment([a_4], [t_1]), assignment([a_4], [t_2]), assignment([a_4], [t_3])} \\
\}
\]
and selects assignment([a_2], [t_1]) (again a nondeterministic choice) and makes
\[
\text{NewObjDescr = [assignment([a_2], [t_1]), assignment([a_1], [t_1])].}
\]
Then
\[
\text{test(assignment([a_2], [t_1]), assignment([a_1], [t_1]), violation)}
\]
fails, hence the next iteration:
\[
\text{test(assignment([a_2], [t_1]), assignment([a_1], [t_1]), finished)}
\]
fails,
\[
\text{propose(assignment([a_2], [t_1]), assignment([a_1], [t_1]), NewObjDescr)}
\]
generates:
\[
\{ \quad \text{assignment([a_3], [t_1]), assignment([a_3], [t_2]), assignment([a_3], [t_3])} \\
\text{assignment([a_4], [t_1]), assignment([a_4], [t_2]), assignment([a_4], [t_3])} \\
\}
\]
and selects assignment([a_3], [t_1]) (again non-deterministically) and makes
\[
\text{NewObjDescr = [assignment([a_3], [t_1]), assignment([a_2], [t_1]), assignment([a_1], [t_1])].}
\]
Now the violation test succeeds because a_3 is not scheduled before a_2:
\[
\text{test([assignment([a_3], [t_1]), assignment([a_2], [t_1]), assignment([a_1], [t_1])], violation)}
\]
leads to a conjunction containing
\[
\neg (\text{DomainRequirement} \vdash a_3 @ t_1 \land a_2 @ t_1 \land a_1 @ t_1 \rightarrow \bot)
\]
which fails. Then revise leads to:
Culprit = culprit([assignment([a_3],[t_1]),
                    assignment([a_2],[t_1]),
                    assignment([a_1],[t_1]),
                    assignment([a_3],[t_1])])
RevisedObjDescr = [assignment([a_2],[t_1]),assignment([a_1],[t_1])] 

In the next iteration:

test([assignment([a_2],[t_1]),assignment([a_1],[t_1])], finished)

fails, and propose generates:

{ assignment([a_4],[t_1]), assignment([a_3],[t_2]), assignment([a_3],[t_3]),
  assignment([a_4],[t_2]), assignment([a_4],[t_3])}

and selects assignment([a_4],[t_1]). This makes

NewObjDescr = [assignment([a_4],[t_1]),
                assignment([a_2],[t_1]),
                assignment([a_1],[t_1])] 

The test for violation succeeds again, and revise leads to:

Culprit = culprit([assignment([a_4],[t_1]),
                    assignment([a_2],[t_1]),
                    assignment([a_1],[t_1]),
                    assignment([a_4],[t_1])])
RevisedObjDescr = [assignment([a_2],[t_1]),assignment([a_1],[t_1])] 

In the next iteration:

test([assignment([a_2],[t_1]),assignment([a_1],[t_1])], finished)

fails. Now propose generates:

{ assignment([a_3],[t_2]), assignment([a_3],[t_3]),
  assignment([a_4],[t_2]), assignment([a_4],[t_3])}

and selects assignment([a_3],[t_2]), and

NewObjDescr = [assignment([a_3],[t_2]),[a_2],[t_1]), assignment([a_1],[t_1])] 

This path eventually leads to a failure to derive a solution. Other choices at the above non-deterministic operations do lead to a solution.