

Structure-preserving specification languages for knowledge-based systems

Frank van Harmelen
Manfred Aben
SWI
University of Amsterdam
{frankh,manfred}@swi.psy.uva.nl

March 28, 1995

Abstract

Much of the work on validation and verification of knowledge based systems (KBSS) has been done in terms of implementation languages (mostly rule-based languages). Recent papers have argued that it is advantageous to do validation and verification in terms of a more abstract and formal specification of the system. However, constructing such formal specifications is a difficult task.

This paper proposes the use of formal specification languages for KBS-development that are closely based on the structure of informal knowledge-models. The use of such formal languages has as advantages that (i) we can give strong support for the construction of a formal specification, namely on the basis of the informal description of the system; and (ii) we can use the structural correspondence to verify that the formal specification does indeed capture the informally stated requirements.

This paper has been submitted to the Journal of Human Computer Studies (formerly the Journal of Man Machine Studies).

1 Introduction

Much of the current research in validation and verification (V&V) of knowledge-based systems (KBS) is done in terms of implementation languages, and mostly rule-based languages [Preece *et al.*, 1992]. State of the art V&V systems check for such properties as loop-freeness of rule-bases, redundancy and eachability of rules, consistency etc. However, it is well known from software engineering, and it is argued in more recent work on V&V of KBS [Vermesan & Wergeland, 1994b, Vermesan & Wergeland, 1994a], that advantages can be gained by doing V&V in terms of a more abstract formal specification language, instead of an implementation language. If we are using an implementation-independent formal specification language, we can (i) do V&V in a much earlier stage of the life-cycle: we can already verify properties of the system in terms of its specification, before the expensive effort of making an implementation, and (ii) we can separate the verification of functional, implementation-independent properties from the verification of implementation-specific properties. Finally, (iii) V&V is facilitated by the

higher level of mathematical abstraction that can be employed in an implementation-independent specification language. In order to obtain these advantages, we must overcome the difficulties associated with formal specification languages. Among these are the large effort involved in creating a formal specification, and the gap that exists between a formal specification language and the informal requirements against which it must be validated.

The claim of this paper is that a close integration of formal and informal descriptions of a KBS is a large step towards solving both of these difficulties. In this paper, we describe a KBS specification language whose structure closely mirrors that of the particular informal description methods used in the earlier stages of the development process. Such close correspondence between formal and informal methods allows us to provide a very strong and even automated support for the construction of a formal specification, and provides strong guidance in the verification and validation of properties of the specification. We call such specification languages “structure preserving specification languages” because they preserve the structure of the informal description in the formal specification. The use of such structure preserving specification languages gives rise to a two-stage development process for KBS: a first stage that is concerned with constructing and refining the specification, and a second stage that is concerned with design and implementation. Since this paper deals with the first stage, we will distinguish a number of substages as follows:

- **Specification:**

1. **Informal model construction:** Informally stated requirements are captured in an informal model. Although informal, such models often have a strong prescribed structure intended to give a conceptual description of the knowledge and problem-solving competence of an agent, be it human expert or KBS. In Sec. 2, we describe the structure of a particular choice for such structured informal models, namely the KADS model of expertise. The initial capturing and subsequent refinement of such structured informal models has been well investigated in the Knowledge Acquisition literature (see [Karbach *et al.*, 1990] for an overview and comparison).
2. **Initial specification construction:** In a second step the informal model is transformed into a fully formal specification of the system. As announced above, this step can greatly benefit from the structure-preserving nature of a formal specification language. In Sec. 3 we describe $(ML)^2$, a formal language closely based on the structure of informally stated KADS expertise models, and in Sec. 4 we show how the structure-preserving nature of $(ML)^2$ can be exploited to generate semi-automatically a formal model from an informal one.
3. **Verification and refinement:** In subsequent steps, the initial formal specification is used for the verification¹ of desired properties, and, if necessary, subjected to further refinement. Section 5 of this paper shows how both the verification of properties and the refinement process are guided by the structural correspondence between formal and informal model.

- **Design and implementation:** Finally, the formal specification will be used as the basis for

¹We are aware of the different meanings assigned to the terms verification and validation in the V&V community [Hoppe & Meseguer, 1991]. Wherever we write “verification”, the reader should interpret this as “verification of specific properties of the specification”. Examples of such properties will be discussed in subsequent sections.

program design and implementation. Although important, this step is not discussed in this paper.

Sections 6 and 7 report on the required software support for the above steps and on our experiences to date with applying these ideas in practice.

As stated above, step 1 is by now well reported in the Knowledge Acquisition literature, and is not an original contribution of this paper. The contributions of this paper are concerned with steps 2 and 3.

To conclude this introduction, we refer to two overview papers from the V&V community to emphasise the relation between this work and the goals of V&V. [Lydiard, 1992] states that “some work addresses how to produce a better specification, and other work concentrates on automated tools for code-checking”, and argues that the work on code-checking has attracted more attention. Our work concentrates on “producing a better specification”.

Perhaps the best way of summarising the value of our work is in terms of [Lydiard, 1992] and [Preece, personal communication]: formalising an informal knowledge -model by using structured specification languages leads naturally and in a principled manner to both “building the right system” and a “building the system right”.

2 Choosing a conceptual structure

The central claim of this paper is that specification languages which are based on a structured conceptual model provide a good medium for system specification and the ensuing validation and verification. In order to substantiate this claim, we are of course under the obligation to provide such a conceptual model. In this section we present such a model. We first present the general structure of a model that is emerging from a number of different areas in AI, and we subsequently describe in more detail a particular approach from Knowledge Engineering that is based on this general model.

The conceptual model we present distinguishes three types of knowledge, and prescribes specific relations between these knowledge-types.

- The first category of knowledge, that we shall call the *domain knowledge* concerns declarative, domain-specific knowledge. Such knowledge describes the objects of discourse in a particular domain, facts that hold about such objects, and relationships among them. This type of knowledge is often represented by rules, facts, hierarchies, objects, properties, relations, etc. A crucial property of this first category of knowledge is that it is represented as much as possible independent from how it will be used. Thus, we state which properties and relations hold in a particular domain, but we do not state how these properties and relations will be used in the reasoning process. That is the concern of the second category of knowledge.
- This second category of knowledge is called *inference knowledge*. Here, we specify (*i*) what the legal inference steps are that we can use in the reasoning process; (*ii*) which role the domain knowledge plays in these inference steps, and (*iii*) what the dependencies are between these inference steps. Again, a crucial property of this type of knowledge is what it does not contain:

although we specify what the legal inference steps are, we do not specify the sequence in which these steps should be applied.

- This sequence of inference steps is exactly the concern of the third type of knowledge, the *procedural knowledge*. This specifies in which order the inferences from the second category should be executed. This type of procedural knowledge is concerned with actions, sequences, iterations, state-transitions, etc.

It is reassuring that examples of the above knowledge-types can be found in widely different sub-fields of symbolic AI. Even a foundational field like logic distinguishes these types of knowledge: domain-knowledge is represented by the axioms of a logical theory, and inference knowledge is embodied in the inference rules of a logic: axioms state what is declaratively true without stating how to use this knowledge, and inference rules describe how to use the domain knowledge in a reasoning process. Logic has traditionally no representation of procedural knowledge, but theorem-proving strategies of automated theorem-provers would correspond to this type of knowledge. Similarly, in planning, we see the same typology: the domain knowledge consists of properties of, say, a blocks world; inference knowledge states the basic planning operations (e.g. *move-on-top-of*), and procedural knowledge concerns strategies for plan formation. Finally, further weight is added to our argument about the general validity of the KADS conceptual model by the observation that modern approaches in object-oriented software engineering such as OMT [Rumbaugh *et al.*, 1991] discern the same three viewpoints on a system: the object model, the functional model and the dynamic model.

Since this paper is concerned with Knowledge Based Systems, we now focus on a particular way of modelling KBSS that is based on the general knowledge types described above. We describe the conceptual models as developed in the KADS project [Wielinga *et al.*, 1993], where they are called “expertise models”. These expertise models will then form the basis for the further arguments in this paper concerning specification languages based on structured conceptual models. The KADS expertise models follow rather closely the 3-layer structure described above.

- The “*domain layer*” of KADS corresponds exactly to the domain knowledge described above. In KADS, this is usually expressed in terms of a KL-ONE-like ontology (concepts, relations, properties, values), although other conceptualisations can be defined.
- KADS inference-layers are often depicted by “*inference structures*” (see Fig. 1), and consist of the following items: (i) “*inference steps*” (indicated by ovals) which are inference steps that map a number of inputs to an output according to the definition of the body of the step (ii) “*knowledge roles*” (indicated by boxes) which constitute the inputs and outputs of the inference steps, and which indicate the role that domain knowledge plays in the inference steps. The connections among the elements of an inference structure capture the dependencies between the inference steps. For example in Fig. 1, the inference step “abstract” produces an output role “observations” which in turn is an input-role for the inference step “hypothesise”. Whereas Fig. 1 gives a graphical representation of an inference structure, Fig. 2 shows part of the same inference structure in a textual format.

The inference structure from Fig. 1 shows part of a simplified diagnostic reasoning process. Using abstraction rules, data about a given patient are translated into more abstract observations. Subsequently, causal rules are used to construct a hypothesis on the basis of these observations.

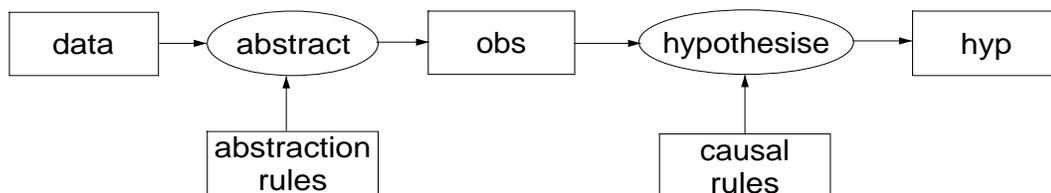


Figure 1: KADS inference layer as a diagram

inference	<i>abstracts</i>
input-roles	<i>data</i> → <i>patient-data</i> ; <i>abstraction-rules</i> → <i>symptom-hierarchy</i> ;
output-roles	<i>obs</i> ;
spec	“Abstract the data into observations”;
inference	<i>hypothesise</i>
input-roles	<i>obs</i> ; <i>causal-rules</i> → <i>symptomatology</i> ;
output-roles	<i>disease</i> ;
spec	“Find a possible cause that explains the observations”;

Figure 2: KADS inference layer

• Notice that Fig. 1 still leaves a lot of freedom for how we apply the two inference steps. This procedural knowledge is stated at the KADS “task layer”. In our example, we might first abstract all data into observations before constructing a hypothesis, or we might construct a hypothesis as soon as we have made the first abstraction. Besides expressing this procedural knowledge, a KADS task-layer also provides a decomposition of the overall task of the system into subtasks. This decomposition has the inference steps from the inference layer as its most primitive elements, and control is expressed among these inference steps.

The three KADS layers and their relationships are summarised in Fig. 3.

In the remainder of this paper we use both the structure of the KADS expertise model and the diagnostic example given above as the basis for further arguments concerning structured specification languages. We claim that the KADS expertise model is a good one: we have shown how its structure is based on a consensus on knowledge types across different fields, and [Wielinga *et al.*, 1993] (and references therein) give ample evidence for the practical utility of KADS. However, KADS is by no means the only possible structure for a conceptual model of KBSS. Different but related models are proposed in [Steels, 1990] and [Chandrasekaran, 1986], and although we base our further arguments on the structure of KADS models, we believe our general arguments to be also valid for these other conceptual models. See for instance our experience with the PROTÉGÉ models, reported in Sec. 7.

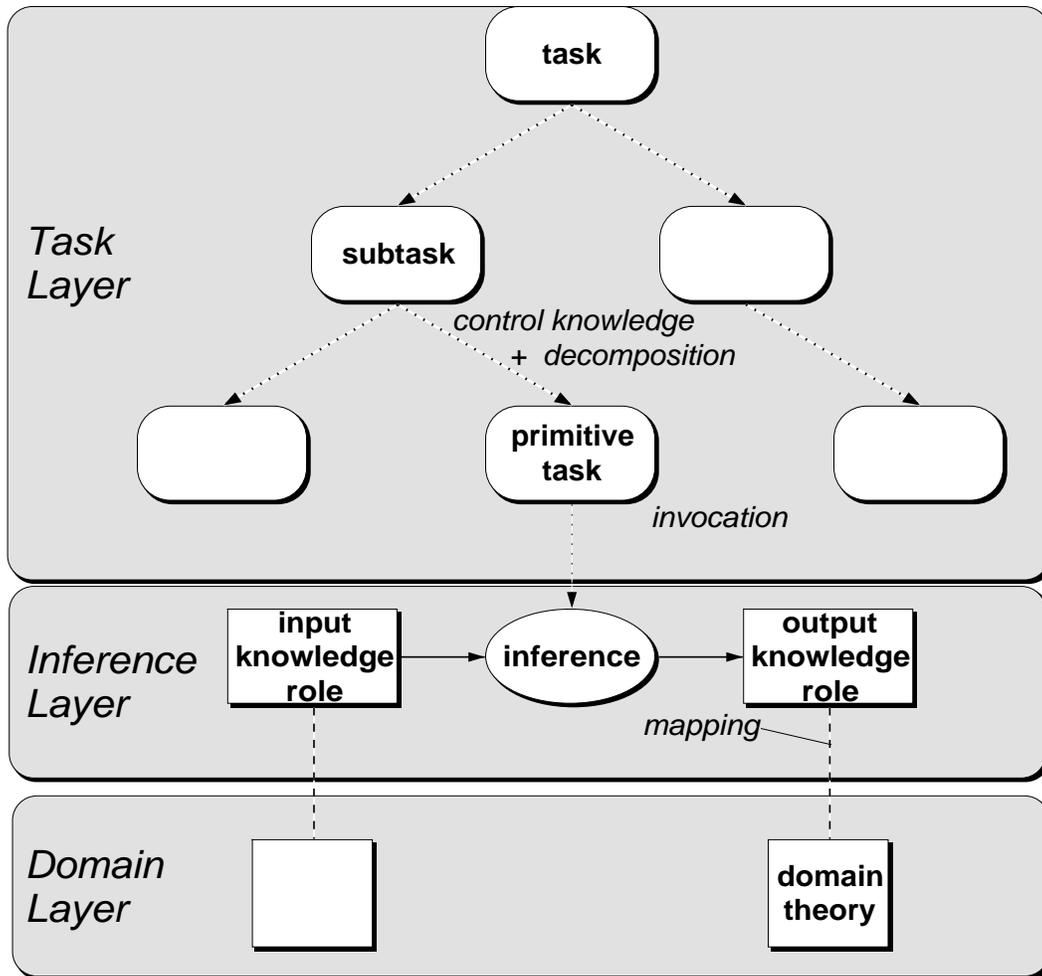


Figure 3: KADS three layer model

3 Formalising such a conceptual structure

If we want to use the above conceptual model for specification and verification purposes, we need a more formal representation of it than the informal descriptions and diagrams given in the previous section. Such a need for more formal representations of conceptual models was widely felt in the Knowledge Engineering community around 1990, and a number of languages that aimed at formalising KADS emerged: KARL [Fensel *et al.*, 1991], MODEL-K [Karbach *et al.*, 1991], K_{BSSF} [Jonker & Spee, 1992], FORKADS [Wetter, 1990]. These and other languages are compared in [Fensel & vanHarmelen, 1994]. In this section we describe our own work in this direction, namely the language (ML)². The formal constructs of (ML)² are the basis for further sections in this paper, but many of these results could also be stated for other KADS languages, or even for KBS specification languages based on different conceptual models such as DESIRE [vanLangevelde *et al.*, 1993] or GETFOL [Giunchiglia *et al.*, 1993].

Our description only states the essential properties of (ML)². For a more detailed description, see

```

theory      patient-data
sort       patient, number
func      temp : patient → number
const     patient1 : patient
axioms    temp(patient1) = 38

theory      symptom-hierarchy
import    patient-data
pred     fever : patient
axioms    $\forall P : temp(P) > 37 \rightarrow fever(P)$ 

theory      symptomatology
import    patient-data, symptom-hierarchy
pred     hepatitis : patient
axioms    $\forall P : hepatitis(P) \rightarrow fever(P)$ 

```

Figure 4: (ML)² domain layer

[vanHarmelen & Balder, 1992]. The foundation for (ML)² is logic, or rather a collection of logics. We will describe how (ML)² formalise s each of the three layers of a KADS expertise model, using the example given in the previous section.

In (ML)², the domain layer is represented in first-order predicate logic (FOPL), although an extension of (ML)² to any two-valued logic with a monotonic deduction relation, e.g. any modal logic, is straightforward. As an additional structuring primitive, (ML)² contains ordered sorts and subtheories. Both are conservative extensions of FOPL. Each (ML)² theory consists of a signature to declare sorts, constants, function- and predicate-symbols and of a set of axioms to describe the domain knowledge. One theory can *import* another, by which the importing theory contains both the signature and the axioms of the imported theory. The domain layer of (ML)² could be summarised by the pseudo-equation:

$$(ML)^2 \text{ domain layer} = \text{FOPL} + \text{ordered sorts} + \text{theories}$$

A simple example of a domain layer is shown in Fig. 4: the signature defines a language for medical knowledge, and three theories are shown; one to represent data about a particular patient, one to represent rules about how raw data like temperature is related to medical terminology like fever, and one to describe medical knowledge about cause-effect relations between diseases and symptoms.

Since a KADS inference layer is “about” the domain knowledge (it specifies the potential use of domain knowledge in the inference process), an inference layer in (ML)² is formalised as a meta-theory of an (ML)² domain layer. Although it is speaking about the sentences of an object-language, a meta-language can remain first-order by introducing terms that are names for object-language expressions. In traditional meta-logics, such names are typically constructed either by quoting object-level expressions, in which case the resulting names have no internal structure (they are regarded as constants of the meta-language), or by mirroring the syntactic structure of the object-

```

role      abstraction-rules
from     symptom-hierarchy
rule      $(P_1 \rightarrow P_2) \mapsto \text{abstrule}(\text{data}(P_1), \text{obs}(P_2))$ 

role      causal-rules
from     symptomatology
rule      $(P_1 \rightarrow P_2) \mapsto \text{causes}(\text{hyp}(P_1), \text{obs}(P_2))$ 

theory   abstract
import   data, abstraction-rules
export   obs
axioms    $\forall X, Y, R : \text{ask-domain-axiom}(\text{data}(X)) \wedge$ 
               $\text{ask-domain-axiom}(R) \wedge R = \text{abstrule}(\text{data}(X), \text{obs}(Y))$ 
               $\leftrightarrow \text{abstract}(\text{data}(X), R, \text{obs}(Y))$ 

theory   hypothesise
import   obs, causal-rules
export   hyp
axioms    $\forall X, Y, R : \text{ask-role}(\text{obs}(X)) \wedge$ 
               $\text{ask-domain-axiom}(R) \wedge R = \text{causes}(\text{hyp}(Y), \text{obs}(X))$ 
               $\leftrightarrow \text{hypothesise}(\text{obs}(X), R, \text{hyp}(Y))$ 

```

Figure 5: (ML)² inference layer

level sentences. In both cases, syntactically similar object-level sentences have syntactically similar names. In contrast to this, names in (ML)² are defined to capture not only the syntax but also the use of the object-level sentences: names are used to represent the knowledge roles in an expertise model. Such names are then constructed by means of rewrite rules that rewrite the object-level expression into a meta-level name that expresses the role that the object-level expression will play in the inference process (see [vanHarmelen, 1992] for more technical details on this).

In the example of an (ML)² inference layer in Fig. 5 contains two such definitions of knowledge-roles: The first knowledge role specifies that object-level implications $P_1 \rightarrow P_2$ from the domain-layer theory *symptom-hierarchy* are named as abstraction rules that map raw data onto observations: $\text{abstrule}(\text{data}(P_1), \text{obs}(P_2))$. The second knowledge role states that implications from another domain layer theory (*symptomatology*) must instead be named as causal rules that link a disease P_1 with the observation P_2 that it causes: $\text{causes}(\text{hyp}(P_1), \text{obs}(P_2))$. It is important to note that such definable names enable us to make distinctions at the inference layer among expressions that are syntactically similar on the domain layer: implications from one domain theory are named as abstraction rules, while implications from another domain theory are named as causal rules. These different names can now be exploited to use these otherwise similar domain-layer expressions in different roles in the inference process.

Besides the knowledge roles, the main constituent of a KADS inference layer are the inference steps. In $(ML)^2$, these are modelled as a predicate whose arguments correspond to the input and output roles of the inference step. The relation that must hold between these input and output roles is defined in a theory for each inference step, and is called the body of the inference step. Figure 5 shows two inference steps: the first applies an abstraction rule to transform input data into abstract observations, while the second applies a causal rule to such an abstract observation in order to construct a hypothesis. Notice that the two types of domain implications (abstraction rules and causal rules) are used in very different ways: abstraction rules are used deductively to derive an abstract observation, while causal rules are used abductively to produce a hypothesis.

To enable the inference layer to inspect the domain layer about which it reasons, $(ML)^2$ contains two special reflective predicates: the predicate *ask-domain-axiom* is true precisely when its argument is the name of an axiom of the domain layer, and the predicate *ask-domain-theorem*, not shown in Fig. 5, is true whenever its argument is the name of a theorem of the domain layer, i.e. it can be deduced from the axioms using rules of first-order logic. These predicates can be used in the body of an inference step. All of this can be summarised in the following pseudo-equation:

$$(ML)^2 \text{ inference layer} = \text{meta-logic} + \text{definable names} + \text{reflective predicates}$$

Because the task layer enforces control over the inference steps, it can be formalised as a program where the primitive statements are the inference steps and knowledge roles are variables upon which these programs operate. In $(ML)^2$ we use the regular program expressions from dynamic logic [Harel, 1984] to formalise such programs. This enables us to specify an imperative language containing sequences, loops and conditional expressions over inference steps. Because the task layer plays no role in any of the examples in this paper, we do not describe it in any further detail.

4 Generating the formal specification from the informal model

As we have seen above, the KADS expertise model is informal, but structured. The formal language $(ML)^2$ follows this structure closely, by providing formal constructs for the elements in the informal model. In this section we show how we can exploit this similarity to generate semi-automatically a formal $(ML)^2$ specification from an informal model.

4.1 Guidelines

When constructing a formal specification from an informal model, good use can be made of the structural correspondence between the formal and informal languages. We have developed a set of guidelines that prescribe how a formal specification should be constructed given an informal model. The guidelines consist of two groups. The majority of the guidelines prescribe how the structure of the informal model should be mapped onto the structure of the $(ML)^2$ specification. A second group of guidelines suggest ways of adding detail to the formal specification on the basis of the informal expertise model.

As we have seen in Sec. 3, $(ML)^2$ provides formal structures for all constructions in a KADS expertise model. This correspondence reduces the number of alternative ways to formalize the expertise

model. For instance, for each inference in the inference layer, an $(ML)^2$ predicate and theory must be defined. Similarly, for each role, a set of rewrite rules is defined. The connections between roles and inferences are formalized by import relations. However, $(ML)^2$ defines additional structure where the expertise model is unstructured. For instance, the actual definition of domain axioms and of inferences are given in unstructured text in the expertise model (Fig. 2), whereas they are formalized as FOPL axioms in $(ML)^2$. Still, the structure of the expertise model can be used to generate a substantial part of the formal specification. For instance, given an inference structure, the arity and type-scheme of the inference predicates can be generated. We can define many such guidelines, some of which provide a limited number of options from which the knowledge engineer can choose. Each such choice effects the possible options in other guidelines. As an example, we can define the following guidelines for generating the structure of the inference layer. In Fig. 6, we repeat the inference structure of Fig. 5, and have underlined all the $(ML)^2$ text that can be generated by using the following selection from our guidelines:

1. Define roles, inference steps and their import relations. These are known from the graphical description of the inference structure (lines 1,4,7–9,13–15)).
2. Determine domain-references for each role (lines 2,5).
3. Define a type for each role. This can be deduced from the role name (lines 3,6).
4. Define a predicate for each inference. The arity and types are known (lines 12,18).
5. Define a general scheme for predicate definition. The I/O connections are known (lines 10–11,16–17).

The full set of sixty-five guidelines is given in [Aben *et al.*, 1994].

4.2 Transformation Tool

The guidelines can to a large degree be automated. Since the formal specification provides more structure than the informal model, the guidelines provide a one-to-many mapping. For instance, the reflective predicate that is used by an inference predicate to refer to the domain knowledge through the roles, may be either *ask-domain-axiom* or *ask-domain-theorem*. Which of these predicates is selected cannot be determined on the basis of the informal model alone, so the user has to make a choice. Many of the guidelines suggest a number of default formalizations, and decisions that are taken in some guidelines determine the decisions that are to be taken in other guidelines. An automated tool can easily keep track of these dependencies, can avoid errors introduced by manually applying the guidelines and can help to preview the results.

We have developed a software tool [Aben *et al.*, 1994] that realizes this functionality. It reduces the number of errors introduced by applying the guidelines manually. This transformation tool, takes as input an informal expertise model, and interactively generates an initial formal specification in $(ML)^2$. For those guidelines that suggest a number of alternative formalizations, the tool prompts for a selection by the user. A number of such selections propagate to other guidelines. For instance, once the types of the roles have been established, there is no need to ask for the types of the arguments to the inference predicates. The tool navigates through the guidelines to minimize the user-interaction. The tool presents sensible default choices. For instance, for roles that have a plural name, such as *observations*, the tool suggests that their type is a set of the singular form,

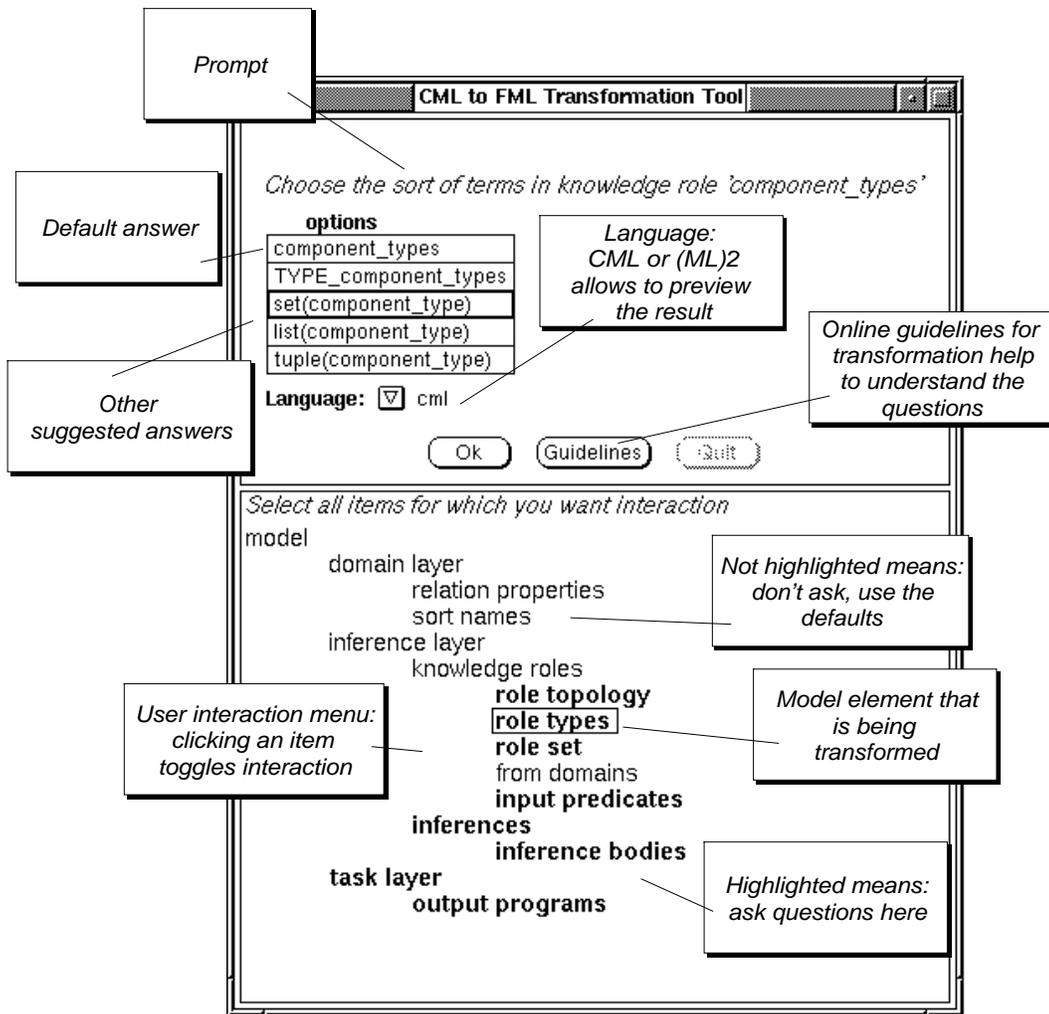


Figure 7: Tool for generating formal specification from informal model

5 Using the formal specification for verification and refinement

Once we have an initial formal specification, as generated by the process described in the previous section, we need to complete the specification and to use it for verification and validation. This means that we want to establish properties of the specified system and check these against our requirements and intentions. In this process, it is again important that there is a structural correspondence between the informal and the formal model. This is important because the requirements on the system are often stated in terms of the informal model, e.g. because they are furnished by the domain expert or the intended end-user, whereas it is the formal specification that is used to prove whether or not the system satisfies these requirements. We must therefore translate the informally stated requirements in terms of the formal specification, and this process is greatly facilitated by a strong structural correspondence between the two. In a similar vein, if it turns out that some requirements are not satisfied by the formal specification, the specification will have to be adjusted accordingly. This adjustment must be guided by and communicated to the

domain-expert or end-user through the corresponding adjustments in to the informal model, and again this hinges on a strong structural correspondence between formal and informal model. Thus, both during verification and during any subsequent refinement, the structure preserving nature of a specification language is important.

This section substantiates this claim in the following ways: in Sec. 5.1 we show how individual components of a specification can be characterised. Section 5.2 uses these characterisations to describe refinement relations between these components. Section 5.3 shows how these components and their relationships can be organised in a library. Section 5.4 describes how individual specification components can be combined to form a full specification, and finally, and perhaps most importantly, Sec. 5.5 shows how we can prove properties of such specifications, and illustrates how we can refine the formal specifications if some intended properties fail to hold.

5.1 Specifications of individual inferences

After the semi-automated generation phase discussed in the previous section, the specification of individual inferences needs to be completed. Section 3 showed that inferences are formalised as predicates and roles as terms that are arguments of these predicates. We can now characterise each inference-predicate as a *pre- and post-condition pair*. Each inference INF with input roles \vec{I} ² and output roles \vec{O} is characterised by a formula of the form:

$$\forall \vec{I}, \vec{O} : \text{PRE}(\vec{I}) \wedge \text{INF}(\vec{I}, \vec{O}) \rightarrow \text{POST}(\vec{I}, \vec{O}) \quad (1)$$

Such a formula states that when the preconditions PRE are enforced on the input roles, and when the relation INF (the body of the inference step) holds between input and output roles, then the postconditions POST hold between input and output roles.

A second useful relation between an inference and its preconditions is that the enforcement of the preconditions on the input roles ensures that the inference can indeed be applied to produce values for the output roles:

$$\forall \vec{I} : \text{PRE}(\vec{I}) \rightarrow \exists \vec{O} : \text{INF}(\vec{I}, \vec{O}) \quad (2)$$

These characterisations are of course very close to the well known Hoare-triples from [Hoare, 1969]. The intuition behind them is that PRE characterise when it is allowed to apply an inference by stating sufficient preconditions, and that POST characterises when it is useful to apply the inference, by stating the effects of the inference.

From formula (1) it is clear that in principle, the post-conditions include the entire deductive closure of the preconditions and the inference-body. In practice, we specify only those parts of the potential post-conditions which are either “interesting” properties of the inference-step, or which are somehow indicative of the “purpose” of the inference step. We show an example of such goal-oriented postconditions in Sec. 5.3 below.

Before we proceed with an example of pre- and postconditions, we must remark that we have chosen to give self-contained but very simple examples. The amount of detail required for a more realistic example would extend beyond the scope of this paper. As a result, our examples may

²The notation \vec{x} is shorthand for a sequence of variables x_1, \dots, x_n .

give the impression that they hardly justify the difficult task of building a formal specification. We refer the interested reader to [Aben, 1995, chapter 8] and [Ruiz *et al.*, 1994] for two examples of larger and more realistic use of the specification framework presented here.

We shall now give an example of a characterisation in terms of pre- and postconditions of the *abstract* inference step from Fig. 1 and 5.

In diagnostic reasoning an essential part of the domain knowledge is a model of the object that is under diagnosis, be it a patient or a machine. A distinction is often made between two types of such domain-models, namely models of correct behaviour or models of faulty behaviour. An important property of the *abstract* step is that it preserves the type of the domain-model that it manipulates. In other words, if an input datum describes faulty (resp. correct) behaviour, it will be abstracted into an observation that again describes faulty (resp. correct) behaviour. This preservation property of the *abstract* step can be characterised by a pre-postcondition pair as follows:

$$\begin{aligned}
 \text{Pre} : & T \in \{ \textit{fault-model}, \textit{correct-model} \} \wedge \\
 & \textit{domain-model-type}(\textit{data}(X), T) \\
 \text{Body} : & (\text{as in fig 5}) \leftrightarrow \textit{abstract}(\textit{data}(X), R, \textit{obs}(Y)) \\
 \text{Post} : & \textit{domain-model-type}(\textit{obs}(Y), T') \rightarrow (T = T')
 \end{aligned} \tag{3}$$

Notice that the formulation of such pre- and postconditions assumes a vocabulary such as *domain-model-type* and *fault-model* which does not immediately follow from the specification of the inference step itself. Such a vocabulary must be based on a theory about the particular domain and task, in our example medical diagnosis. Establishing such a theory and the corresponding vocabulary for expressing pre- and postconditions is by no means a trivial task. In this paper, we shall simply assume the existence of such a vocabulary for our examples. In the case of diagnosis, the suitability criteria identified in [Benjamins, 1994] are a good example of an analysis that contributes to such a vocabulary. A further assumption is that the specification language itself (and in particular, the language for pre- and postconditions) is strong enough to express such properties. Our experience indicates that this is indeed the case for the inference layer. In section 8 we will discuss the extensions that are perhaps needed to fulfill this assumption for the task-layer.

5.2 Refinement Calculus

While constructing and refining a model it is useful to know which relationships hold among different model elements, and among different versions of the same model element. If during validation and verification it turns out that a given specification does not satisfy some requirements, we must modify the specification. Such a modification will often be performed by replacing a component of the specification with a different version of that component. It is then important to know what the logical relations are between the original and the new version of the component, because these relations will enable us to select which replacement should be selected.

We now show how for inferences it is possible to construct a *refinement calculus* which establishes such relations. Two examples of such relations among inference steps are weakening and strengthening:

$$\begin{aligned}
 \text{WEAKEN}(\text{INF}_1) & \triangleq \text{INF}_2 \quad \textit{iff} \quad \text{INF}_1 \vdash \text{INF}_2 \\
 \text{STRENGTHEN}(\text{INF}_1) & \triangleq \text{INF}_2 \quad \textit{iff} \quad \text{INF}_2 \vdash \text{INF}_1
 \end{aligned}$$

which state that INF_1 is either more general or more specific than INF_2 .

The following are concrete instances of these general relations, and show how stronger and weaker versions of an inference can be obtained through specific syntactic manipulations:

$$\begin{aligned} \wedge\text{-STRENGTH}_\alpha(\text{INF}) &\stackrel{\Delta}{=} \text{INF} \wedge \alpha \\ \wedge\text{-WEAK}_\alpha(\text{INF} \wedge \alpha) &\stackrel{\Delta}{=} \text{INF} \\ \vee\text{-STRENGTH}_\alpha(\text{INF} \vee \alpha) &\stackrel{\Delta}{=} \text{INF} \\ \vee\text{-WEAK}_\alpha(\text{INF}) &\stackrel{\Delta}{=} \text{INF} \vee \alpha \end{aligned}$$

An example of the use of strengthening is to take the inference step *abstract*, and to apply $\wedge\text{-STR}_\alpha$ to it with for α a uniqueness condition, resulting in the following inference step:

$$\begin{aligned} \forall X, Y, R : \quad &ask\text{-domain-axiom}(data(X)) \wedge \\ &ask\text{-domain-axiom}(R) \wedge R = absrule(data(X), obs(Y)) \wedge \\ &\forall Y' : ask\text{-domain-axiom}(absrule(data(X), obs(Y'))) \rightarrow (Y = Y') \\ &\leftrightarrow unique\text{-abstract}(data(X), R, obs(Y)) \end{aligned}$$

This ensures that we only obtain abstractions when they are uniquely defined, and is clearly a strengthening of *abstract*, since

$$unique\text{-abstract}(data(X), R, obs(Y)) \rightarrow abstract(data(X), R, obs(Y))$$

The above operators (and others described in [Aben, 1995, chapter 7]) have useful algebraic properties:

- each operator has an inverse, e.g:

$$\wedge\text{-WEAK}_\alpha(\wedge\text{-STRENGTH}_\alpha(\text{INF})) = \text{INF}.$$

This is useful since it often allows us to simplify a long chain of strengthenings and weakenings into a shorter one.

- The operators are idempotent, so that repetition of the same operator has no effect:

$$\wedge\text{-STRENGTH}_\alpha(\wedge\text{-STRENGTH}_\alpha(\text{INF})) = \wedge\text{-STRENGTH}_\alpha(\text{INF})$$

- The operators distribute over conjunction and disjunction, e.g:

$$\begin{aligned} \wedge\text{-STRENGTH}_{\alpha \wedge \beta}(\text{INF}_1 \wedge \text{INF}_2) &\leftarrow \wedge\text{-STRENGTH}_\alpha(\text{INF}_1) \wedge \\ &\wedge\text{-STRENGTH}_\beta(\text{INF}_2) \end{aligned}$$

This shows that when we want to strengthen an entire inference structure, expressed by $\text{INF}_1 \wedge \text{INF}_2$, we can proceed by strengthening individual inferences steps in the inference structure.

5.3 A library of inferences: selection and refinement

Much of the work in Knowledge Engineering emphasises the importance of libraries of reusable components out of which the (mostly informal) models can be constructed. All the important Knowledge Engineering methodologies (KADS, Generic Tasks, Components of Expertise), propose such libraries and [Breuker & Van de Velde, 1994] is a recent example of a library of KADS inference steps. Just as for informal models, a library of predefined formal specification components would be an important step towards making formal specifications for KBS a practical possibility. However, two major problems in realising libraries of such specification components are (i) how to index the library so that components can be retrieved when (and only when) appropriate, and (ii) how to organise the contents of the library by providing relations among the components. The use of a structured specification language as presented here offers solutions to both these problems.

First of all, the postconditions of inference steps can be used to index a library, because, as announced above, the postconditions can be used to express the “goal” or “purpose” of an inference step. For example, a goal-oriented postcondition of the *unique-abstract* step defined above could be that *unique-abstract* guarantees not to make search-spaces larger and might make them smaller:

$$\begin{aligned} \forall D, O : \quad & D = \{data(X_1), \dots, data(X_n)\} \wedge \\ & O = \{obs(Y) \mid data(X_i) \in D \wedge \exists R : unique-abstract(data(X_i), R, obs(Y))\} \\ & \rightarrow |O| \leq |D| \end{aligned}$$

Such a postcondition is a useful way to index a library of specification components: if, while building our formal specification, we are looking for inference steps to reduce a search space, searching in the library with a condition as above will guide us to the appropriate inferences in the library. This illustrates how we should carefully formulate the postconditions of an inference as those subsets of its consequences which indicate its purpose and its useful properties.

Obviously, a single inference may serve different purposes, which can be expressed by defining multiple post-conditions for the inference. For example, besides reducing a search space, the *abstract* interface might also be used to map one vocabulary onto another. The expression of such post-conditions again hinges on the availability of a vocabulary and an underlying conceptualisation concerning domain and task to introduce such notions as “size of search space”, for instance.

Whereas the pre- and postconditions allow us to *index* the library, the refinement operators enable us to *relate* the elements of the library: Relations between library elements can be expressed in terms of the refinement operators, for example if we take

$$\alpha = \forall Y' : ask-domain-axiom(abstract(data(X), obs(Y'))) \rightarrow (Y = Y')$$

then:

$$\wedge\text{-STRENGTH}_\alpha(abstract(data(X), R, obs(Y))) = unique-abstract(data(X), R, obs(Y))$$

Together with other strengthenings and weakenings of *abstract*, this organises a library as a graph, where inferences are the nodes and refinement operators are the edges. [Aben, 1995, chapter 5] describes in detail a library of a few dozen inference steps which are specified, indexed and organised in this way.

5.4 Combining inferences

The previous subsections have shown how to specify and organise individual inferences. In order to construct full-scale specifications, we must of course combine such inferences into a coherent inference structure. We can again exploit the structure of our specification language to support this task.

In a KADS inference layer, we must ensure that inferences which are connected by means of a shared role are indeed compatible in some sense. Our formal notion of compatibility of two inferences is defined as follows:

Let INF_1 and INF_2 be two inferences, with preconditions PRE_1 and PRE_2 respectively, let r be an input role of INF_2 , and let PRE_2^r be the subset of the preconditions of INF_2 which are defined on role r , then INF_1 and INF_2 are compatible on role r iff

$$\forall \vec{i}, \vec{o} : \text{PRE}_1(\vec{i}) \wedge \text{INF}_1(\vec{i}, \vec{o}) \wedge (\exists k : r = o_k) \rightarrow \text{PRE}_2^r(r) \quad (4)$$

The intuition behind this definition is that when INF_1 and INF_2 are connected through role r (since one of the output roles o_k of INF_1 is equal to the input role r of INF_2), then the properties we can derive about r as an output role of INF_1 are sufficient to ensure the properties required for r as an input role of INF_2 .

In fact this definition is only sufficient for very simple inference structures as the one in Fig. 1. In [Aben, 1995, chapter 7] we describe how the definition of compatibility can be generalised to deal with multiple input roles of INF_2 , each of which may in turn be the output role of one or more inference steps. Furthermore, a generalised notion of compatibility would not only quantify over all input roles r of INF_2 , but would also allow the proof of PRE_2^r to be not only on the basis of the immediately preceding inference step INF_1 , but also on the basis of all inference steps that lead up to INF_1 . This leads to a definition of compatibility that takes into account the entire inference structure preceding INF_2 .

Finally, the definition above is based on the intuition that INF_1 ensures the preconditions of INF_2 for all the possible values of output role r . A weaker alternative is to only demand that INF_2 's preconditions hold for some values of r :

$$\forall \vec{i} \exists \vec{o} : \text{PRE}_1(\vec{i}) \wedge \text{INF}_1(\vec{i}, \vec{o}) \wedge (\exists k : r = o_k) \rightarrow \text{PRE}_2^r(r)$$

Yet another, and again weaker, alternative notion of compatibility would only demand that the preconditions of INF_2 are at least consistent with the consequences of INF_1 for all or some values of r . The point here is not which of these decreasingly strong notions of compatibility is the right one in all cases, but rather that any of these formal compatibility requirements is closely linked with an underlying intuition in terms of the properties of the informal model. This link then makes it possible to choose the appropriate notion in any given case, depending on the application requirements.

A simple example of using requirement (4) in the context of Fig. 1 is the following. If we know that the causal rules in our application are taken from a fault-model then the *hypothesise* step which uses these causal rules is only valid for observations of faulty behaviour. This could be

expressed in the preconditions of *hypothesise* as;

$$Pre : \text{domain-model-type}(\text{obs}(X), \text{fault-model})$$

In order to prove compatibility of the inference structure from Fig. 1, we must now, following (4), prove that this precondition follows from the preconditions and body of the *abstract* step. Since (3) states that the *abstract* inference preserves the domain-knowledge-type, compatibility now amounts to proving that the input data to *abstract* is indeed of type *fault-model*. This shows how an attempt to verify the compatibility requirement (4) eventually results in a proof obligation on the input data of the system, namely that only data about faulty behaviour is used.

The refinement operators defined above are again useful in this context: if for some reason we fail to prove the compatibility requirement (4) between two inferences INF_1 and INF_2 , we could try to find an additional condition α so that the compatibility between $\wedge\text{-STRENGTH}_\alpha(INF_1)$ and INF_2 does hold.

5.5 Proving properties

The formal apparatus we have described until now consists of inferences characterised as pre- and postconditions, a refinement calculus over these inferences, and a way to combine inferences into inference structures. In this section we show how this apparatus can be used to validate and verify specifications, and to adjust the specifications if necessary. We do this first for an individual inference step, and then for an entire inference structure.

Properties of a single inference step

To illustrate this process, we give the following example. Assume that part of the system requirements state that the abstraction process should be transitive: in other words, the *abstract* inference step should be able to chain together abstraction rules from the domain layer. This required property can be expressed in the following way:

$$\begin{aligned} Pre & : \exists R_1, R_2 : \text{abstract}(X, R_1, Y) \wedge \text{abstract}(Y, R_2, Z) \\ Body & : \text{(as in fig 5)} \\ Post & : \exists R_3 : \text{abstract}(X, R_3, Z) \end{aligned} \tag{5}$$

Unfortunately, the postcondition of (5) is not provable from the stated precondition and the body of the *abstract* step: as it stands, the *abstract* step only considers individual domain-rules, and does not attempt to chain them together. This failure to verify a desired property should now trigger an adjustment in the specification. In this case two adjustments are possible.

A first possibility would be to adjust the body of the *abstract* step to chain together individual domain rules. This would be obtained by replacing the (ML)² predicate *ask-domain-axiom* by the (ML)² predicate *ask-domain-theorem*. Remember that *ask-domain-axiom* checks if its argument is a domain axiom, whereas *ask-domain-theorem* checks if its argument is provable from the domain layer. This would enable the *abstract* step to exploit the transitivity of implication at the domain layer, and thereby exhibit the behaviour required in (5).

A second possibility would be to leave the *abstract* step unmodified, but to ensure that the domain layer itself contains the full transitive closure of all abstraction rules. In that case, we could strengthen the precondition of (5) to

$$\begin{aligned}
 \text{Pre : } & \text{precondition as in (5)} \wedge \\
 & (\text{ask-domain-axiom}(\text{absrule}(\text{data}(X), \text{obs}(Y))) \wedge \\
 & \quad \text{ask-domain-axiom}(\text{absrule}(\text{data}(Y), \text{obs}(Z))) \rightarrow \\
 & \quad \text{ask-domain-axiom}(\text{absrule}(\text{data}(X), \text{obs}(Z))) \\
 &)
 \end{aligned}$$

and the required postcondition of (5) is now provable from this strengthened precondition plus the (unaltered) definition of the *abstract* step.

Notice the important difference between the above two adjustments in response to a failed verification obligation: in the first solution, we changed the definition of an inference step (we strengthened it), while in the second, we changed the contents of the domain theory upon which this step operates.

Properties of an inference structure

We illustrate the verification of an entire inference structure using the following scenario: suppose that the requirements on the system state that a given piece of data d should lead to a particular hypothesis h , and suppose that this requirement is not fulfilled by the system.

In order to use the formal specification to deal with this problem, we introduce the following notation: we write \mathcal{SPEC} to denote the entire specification consisting of the domain layer from Fig. 4 and the inference layer from Fig. 5. Furthermore, let $\text{diagnose}(\text{data}(X), \text{hyp}(Y))$ mean that in the inference structure of Fig. 1, we obtain a hypothesis Y when given a data X . Formally:

$$\text{diagnose}(\text{data}(X), \text{hyp}(Y)) \triangleq \exists R_1, R_2, Z : \text{abstract}(\text{data}(X), R_1, \text{obs}(Z)) \wedge \text{hypothesise}(\text{obs}(Z), R_2, \text{hyp}(Y)). \quad (6)$$

We can then express the violated requirement as the failure to prove:

$$\mathcal{SPEC} \vdash \text{diagnose}(\text{data}(d), \text{hyp}(h)). \quad (7)$$

Using (6) this is equivalent to failing to prove:

$$\mathcal{SPEC} \vdash \exists R_1, R_2, Z : \text{abstract}(\text{data}(d), R_1, \text{obs}(Z)) \wedge \text{hypothesise}(\text{obs}(Z), R_2, \text{hyp}(h)).$$

This failure can be reduced to one of the following three cases:

Case 1: We fail to prove:

$$\mathcal{SPEC} \vdash \exists R_1, Z : \text{abstract}(\text{data}(d), R_1, \text{obs}(Z)) \quad (8)$$

This means that no abstraction rules exist that abstract d into an observation, in other words: d simply does not occur in the data-vocabulary. The only remedy for this is then to add rules which state how d must be abstracted into an observation.

Case 2: We fail to prove:

$$SP\mathcal{E}\mathcal{C} \vdash \exists R_2, Z' : \text{hypothesise}(\text{obs}(Z'), R_2, \text{hyp}(h)) \quad (9)$$

This means that no causal rules exist that mention h , in other words: h does not occur in the hypothesis vocabulary. The remedy for this case is to add rules which state how h might cause an observation.

Case 3: We can prove both (8) and (9), but never for values $Z = Z'$. In other words, the set of observations which are abstractions of d is disjoint from the set of observations which are caused by h :

$$\begin{aligned} \text{Let } OBS_d &= \{ \text{obs}(Z) \mid \exists R_1, Z : \text{abstract}(\text{data}(d), R_1, \text{obs}(Z)) \} \\ \text{and } OBS_h &= \{ \text{obs}(Z') \mid \exists R_2, Z' : \text{hypothesise}(\text{obs}(Z'), R_2, \text{hyp}(h)) \} \\ \text{then } OBS_d \cap OBS_h &= \emptyset. \end{aligned}$$

This suggests that we should add a rule to either of these sets to ensure that this intersection is no longer empty. If we keep the observation vocabulary constant, then we can even enumerate all possible ways of doing this. Let

$$\begin{aligned} RULES = & \{ \text{absrule}(\text{data}(d), \text{obs}(Z') \mid \text{obs}(Z') \in OBS_h \} \cup \\ & \{ \text{causes}(\text{hyp}(h), \text{obs}(Z) \mid \text{obs}(Z) \in OBS_d \}, \end{aligned}$$

then for any rule R from $RULES$:

$$SP\mathcal{E}\mathcal{C} \cup \{R\} \vdash \text{diagnose}(\text{data}(d), \text{hyp}(h)),$$

which is the desired result.

In this scenario, a translation of the initial requirement into a formal statement was followed by an analysis of this formal statement. This yielded three possible reasons for the failure to comply with the requirement: either the data vocabulary was incomplete, or the hypothesis vocabulary was incomplete, or a rule connecting the two vocabularies through an observation was missing. In this latter case, we exploited the formal machinery to enumerate all rules that might be added to the system to fulfill the requirement. This set of possible repairs might then be given to a domain expert who could select the appropriate element of this set. This scenario illustrates how the structure of the formal specification can be exploited in the analysis of a desired property we failed to verify.

The analysis in case 3 above is taken even further in [vanDompsele & vanSomeren, 1994], where Machine Learning techniques are applied to an (ML)² specification in order to automatically suggest possible responses to a violated requirement. This very promising line of research for the verification and validation field becomes possible by exploiting the strong structure of a specification language like (ML)².

6 Software Tool Support

The complexity of creating, reading and evaluating a formal specification requires the support of good software tools. The Si(ML)² tool set [Aben *et al.*, 1994] has been designed to provide support in each phase of the formal specification process.

We briefly discuss the tools in terms of the phases in formal specification mentioned in Sec. 1. These phases were: initial specification construction, specification refinement and specification verification.

Initial specification construction: transformation tool. We have discussed the transformation tool that supports initial specification construction in Sec. 4 above.

Specification refinement: editor. Our $(ML)^2$ -editor provides a number of functions that support browsing and navigating through the specification text. Keywords, comments and identifiers are displayed in different fonts, which makes the specification text more readable, and avoids many typing errors. Different levels of the specification text can be hidden from the display, to provide better overview of the specification. The $(ML)^2$ editor makes it possible to move through the text in terms of the structure of the specification, e.g. in terms of layers or theories in a layer. The editor can also display or jump to the declaration of a symbol on which the cursor stands. Balder [Balder & Akkermans, 1992] presents the graphical editor and parser THEME for $(ML)^2$ specifications. Experiments with integrating THEME and $Si(ML)^2$ give some evidence of the benefits of combining text-oriented interfaces with graphical editors. $Si(ML)^2$ is also incorporated in the CommonKADS knowledge engineering workbench. The workbench provides a graphical front-end to $Si(ML)^2$.

Specification verification: parser, checker and interpreter. For the evaluation phase, $Si(ML)^2$ provides an incremental parser, a type-checker and a simulation tool that acts as an interpreter for a subset of $(ML)^2$. The parsing process can be visualised, which helps in finding the location of syntactical errors. A pretty printer can be used to normalize layout of the specification, and to generate \LaTeX documentation. When a specification has been successfully parsed, the checker makes sure that all referenced symbols are declared and well-typed, and that the import relations are correct. The user-interface makes it possible to jump to the location of the error by a single keystroke. $Si(ML)^2$ includes a simulator tool, that can be used to interpret the specification and to prove properties of it. We rely as much as possible on existing theorem proving technology, rather than reinvent many of the theorem proving wheels. The simulator contains an interactive tracer, that can be used to inspect the intermediate steps.

7 Applications

We have applied the theory and tools discussed in this paper in a number of projects. Furthermore, other research groups have applied our results in their projects. Although these projects have until now been small, it gives us enough confidence in our current results to continue our research along these lines, and to apply the results to larger projects. In this section we briefly describe our experience to date.

Our formalism has been used by ourselves and others for specification in the following areas: temporal abstraction [Aben, 1995, chapter 6], problem solving methods in diagnosis [Aben, 1995, chapter 8], a simple scheduling method [Balder *et al.*, 1993], analysing underwater sonar data [Wols, 1993, van 'tHolt, 1993], and safety specification in medical applications [Fox, 1993].

In [Ruiz *et al.*, 1994] and [Aben, 1995, chapter 8] we have analysed the advantages that the use of

a formal specification framework as presented here has brought to the modelling of a scheduling algorithm and of a set of diagnostic methods.

In [Aben, 1995] we have taken a subset of the problem solving methods for diagnosis as described in [Benjamins, 1993], with the following results:

- a number of inconsistencies in the informal descriptions were located;
- hidden assumptions in the informal descriptions were uncovered (e.g. about the completeness of available knowledge);
- implicit dependencies between inference steps were revealed;
- we were able to verify formally the informally stated claims about the diagnostic methods.

In [Ruiz *et al.*, 1994] we have reported on an in-house evaluation study of both the (ML)² language and the Si(ML)² toolset. We analysed the types of errors in an informal model that are revealed during formalisation of a model of a scheduling task in (ML)². These error-types were:

- missing dependencies between inference steps;
- inferences that had been given the wrong name;
- inclusion of control knowledge in the inference layer;
- widely varying grainsize of inference steps;
- redundancies between parts of the model.

8 Discussion

A major recent survey of formal methods in Software Engineering, undertaken by the USA Department of Commerce [Craig *et al.*, 1993], concluded that a stronger integration with informal methods was the main prerequisite for an increased usefulness of formal specification languages. In this paper, we have shown how such a strong integration can be achieved for KBS. We have taken a particular informal conceptual model of a KBS structure, and showed how to define a formal specification language that closely follows this structure. We have also shown what the benefits are of such a structure-preserving formal language:

- The structure of the informal model can be used to generate large parts of the formal specification semi-automatically.
- The formal specification can be divided into meaningful components, and each of these components can be characterised by properties that can be used to express relations between them.
- A refinement calculus can be defined between different versions of a component. Such a calculus can be used to organise a library of components.
- Compatibility requirements can be formulated which ensure that components are combined in a proper way to form a coherent specification.

- The structure-preserving nature of the formal language enables us to translate informally stated requirements into formally verifiable statements.
- When a requirement is not fulfilled, the structure of the specification into components helps to find the required adjustments to the specification.

We have also shown how such a specification language can be supported through appropriate software tools. Some of these tools are part of the standard repertoire of tools in Software Engineering (editor, type-checker etc), but the tool that semi-automatically generates parts of the formal specification from the informal model goes beyond the functionality of traditional software support.

Future work: Although we claim that the work presented here is a significant step towards the use of formal specification languages in KBS development, a lot of work still remains to be done.

Our refinement calculus is only descriptive in nature: it shows which refinements of a specification *can* be made. Ultimately, we would like to move towards a prescriptive refinements theory, which states which refinements *should* be made in a given case.

We have shown how to move from an informal to a formal model, but it is currently an open question how adjustments in the formal model should be mapped back to the informal model. Such a mapping back is essential to keep the two models synchronised during the development process.

Most of the examples in this paper concentrated on the inference steps as components of a specification. [Aben, 1995] shows that our ideas can be extended to cover entire sub-graphs of an inference structure. The development of similar ideas for the other components of a specification, such as domain-layer theories or subtasks in the task layer remains open. In particular, for the task-layer, we expect that extensions to the framework presented here will be required. Since the task-layer is concerned with control-knowledge, it has to deal with sequences of actions and a notion of state. First-order logic will not be sufficient to specify these notions. In (ML)², we employ program-expressions from dynamic logic [Harel, 1984] to specify the task layer. In this paper, we have assumed that the language for expressing (parts of) a conceptual model is the same as the language for expressing the pre- and postconditions of these parts. Formulas (1), (2) and (3) relied on this. By moving away from first order logic for specifying parts of the conceptual model, this will no longer be true. For example, instead of formula (1), in dynamic logic we must write:

$$\forall \vec{I}, \vec{\sigma} : \text{PRE}(\vec{I}) \rightarrow [\text{INF}]\text{POST}(\vec{I}, \vec{\sigma}).$$

The full consequence of using such a modal language for the process of verifying and refining conceptual models remain to be investigated.

Besides the tools discussed in Sec. 6, further tool support must be developed for managing the library, for using the refinement calculus, and for proving properties of specifications.

Finally, although we have mentioned in Sec. 7 a number of small to medium-scale applications of our ideas, the confrontation with large-scale industrial applications is still a challenge.

Acknowledgements

We are grateful to our colleagues Anjo Anjewierden, Dieter Fensel, Gertjan van Heijst, Guus Schreiber, Remco Straatman and Annette ten Teije for providing us with useful feedback on an earlier version of this paper.

References

- [Aben, 1995] M. Aben. *Formal Methods in Knowledge Engineering*. PhD thesis, University of Amsterdam, Faculty of Psychology, February 1995. ISBN 90-5470-028-9.
- [Aben *et al.*, 1994] M. Aben, J. Balder, and F.A.H. van Harmelen. Support for the formalisation and validation of KADS expertise models. ESPRIT Project P5248 KADS-II/M2/TR/UvA/DM2.6a/1.0, University of Amsterdam, januari 1994.
- [Balder & Akkermans, 1992] J.R. Balder and J.M. Akkermans. TheME: an environment for building formal KADS-II models of expertise. *AI Communications*, 5(3):136, 1992.
- [Balder *et al.*, 1993] J. Balder, F. van Harmelen, and M. Aben. A KADS/ML² model of a scheduling task. In J. Treur and Th. Wetter, editors, *Formal Specification of Complex Reasoning Systems*. Ellis Horwood, 1993.
- [Benjamins, 1993] V. R. Benjamins. *Problem Solving Methods for Diagnosis*. PhD thesis, University of Amsterdam, Amsterdam, The Netherlands, June 1993.
- [Benjamins, 1994] V. R. Benjamins. On a role of problem solving methods in knowledge acquisition – experiments with diagnostic strategies –. In L. Steels, A. T. Schreiber, and W. van de Velde, editors, *Lecture Notes in Artificial Intelligence, 867, 8th European Knowledge Acquisition Workshop, EKAW-94*, pages 137–157, Berlin, Germany, 1994. Springer-Verlag.
- [Breuker & Van de Velde, 1994] J. A. Breuker and W. Van de Velde, editors. *The CommonKADS Library for Expertise Modelling*. IOS Press, Amsterdam, The Netherlands, 1994.
- [Chandrasekaran, 1986] B. Chandrasekaran. Generic tasks in knowledge based reasoning: High level building blocks for expert system design. *IEEE Expert*, 1(3):23–30, 1986.
- [Craigien *et al.*, 1993] D. Craigien, S. Gerhart, and T. Ralston. An international survey of industrial applications of formal methods. Technical report, U.S. Department of Commerce, Technology administration, National Institute of Standards and Technology, Computer Systems Laboratory, Gaithersburg, MD 20899, USA, March 1993.
- [Fensel & vanHarmelen, 1994] D. Fensel and F. van Harmelen. A comparison of languages which operationalise and formalise KADS models of expertise. *The Knowledge Engineering Review*, 9:105–146, 1994.
- [Fensel *et al.*, 1991] D. Fensel, J. Angele, and D. Landes. Knowledge representation and acquisition language (KARL). In *Proceedings 11th International workshop on expert systems and their applications (Volume: Tools and Techniques)*, pages 821–833, Avignon, France, May 1991.
- [Fox, 1993] J. Fox. On the soundness and safety of expert systems. *Artificial Intelligence in Medicine*, 5:159–179, 1993.
- [Giunchiglia *et al.*, 1993] E. Giunchiglia, P. Traverso, and F. Giunchiglia. Multi-context systems as a specification framework for complex reasoning systems. In J. Treur and Th. Wetter, editors, *Formal Specification of Complex Reasoning Systems*. Ellis Horwood, 1993.

- [Harel, 1984] D. Harel. Dynamic logic. In D. Gabbay and F. Guentner, editors, *Handbook of Philosophical Logic, Vol. II: extensions of Classical Logic*, pages 497–604. Reidel, Dordrecht, The Netherlands, 1984.
- [Hoare, 1969] C.A.R. Hoare. The axiomatic basis of computer programming. *Communications of the ACM*, 12(10):567–583, October 1969.
- [Hoppe & Meseguer, 1991] T. Hoppe and P. Meseguer. On the terminology of VVT. In M. Grisoni, editor, *EUROVAV'91: Proceedings of the European Workshop on the Verification and Validation of Knowledge Based Systems*, pages 103–108, 1991.
- [Jonker & Spee, 1992] W. Jonker and J. W. Spee. Yet another formalisation of KADS conceptual models. In Th. Wetter, K.-D. Althoff, J. Boose, and B. Gaines, editors, *Current developments in knowledge acquisition: EKAW'92*, volume 509 of *Lecture Notes in AI*. Springer-Verlag, 1992.
- [Karbach *et al.*, 1990] Werner Karbach, Marc Linster, and Angi Voß. Models, methods, roles and tasks: Many labels - one idea? *Knowledge Acquisition*, 2(4):279–300, 1990.
- [Karbach *et al.*, 1991] W. Karbach, A. Voß, R. Schukey, and U. Drouwen. Model-K: Prototyping at the knowledge level. In *Proceedings Expert Systems-91*, pages 501–512, Avignon, France, 1991.
- [Lydiard, 1992] T. Lydiard. Overview of current practice and research initiatives of the verification and validation of KBS. *Knowledge Engineering Review*, 7(2):101–113, 1992.
- [Preece *et al.*, 1992] A. Preece, R. Shinghal, and A. Batarekh. Principles and practice in verifying rule-based systems. *Knowledge Engineering Review*, 7(2):115–141, 1992.
- [Ruiz *et al.*, 1994] F. Ruiz, F. van Harmelen, M. Aben, and J. van de Plassche. Evaluating a formal specification language. In L. Steels, A.Th. Schreiber, and W. Van de Velde, editors, *A Future for Knowledge Acquisition, Proc. 8th EKAW*, number 867 in *Lecture Notes in Artificial Intelligence*, pages 26–45. Springer-Verlag, 1994.
- [Rumbaugh *et al.*, 1991] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorenzen. *Object-Oriented Modelling and Design*. Prentice Hall, Englewood Cliffs, New Jersey, 1991.
- [Steels, 1990] L. Steels. Components of expertise. *AI Magazine*, Summer 1990.
- [van 'tHolt, 1993] M. van 't Holt. Modelling of visual perception for a recognition task in noise analysis. Master's thesis, Dept. of Information Theory, Fac. of Electrical Engineering, Technical Univ. of Delft, August 1993. (in Dutch).
- [vanDompsele & vanSomeren, 1994] H. van Dompsele and M. van Someren. Using models of problem solving as bias in automated knowledge acquisition. In *Proceedings of the 11th European Conference on AI, ECAI'94*, pages 503–507, Amsterdam, August 1994. Wiley.
- [vanHarmelen & Balder, 1992] F. van Harmelen and J. R. Balder. (ML)²: a formal language for KADS models of expertise. *Knowledge Acquisition*, 4(1), 1992. Special issue: 'The KADS approach to knowledge engineering', reprinted in *KADS: A Principled Approach to Knowledge-Based System Development*, 1993, Schreiber, A.Th. *et al.* (eds.).
- [vanHarmelen, 1992] F. van Harmelen. Definable naming relations in meta-level systems. In A. Pettorossi, editor, *Proceedings of the Third Workshop on Meta-programming in Logic (META'92)*, volume 649 of *Lecture Notes in Computer Science*, pages 89–104, Uppsala, June 1992. Springer-Verlag.
- [vanLangevelde *et al.*, 1993] I. van Langevelde, A. Philipsen, and J. Treur. A compositional architecture for simple design formally specified in DESIRE. In J. Treur and Th. Wetter, editors, *Formal Specification of Complex Reasoning Systems*. Ellis Horwood, 1993.
- [Vermesan & Wergeland, 1994a] A. Vermesan and T. Wergeland. Expert system verification and validation. Working Paper 92/1994, Centre for Research in Economics and Administration SNF, Bergen, Norway, July 1994.

- [Vermesan & Wergeland, 1994b] A. Vermesan and T. Wergeland. A formally-based methodology for deriving verifiable expert systems from specifications. In *Proceedings of the AAAI'94 workshop on validation and verification of knowledge-based systems*, Seattle, July 1994.
- [Wetter, 1990] T. Wetter. First-order logic foundation of the KADS conceptual model. In B. J. Wielinga, J. Boose, B. Gaines, G. Schreiber, and M. van Someren, editors, *Current trends in knowledge acquisition*, pages 356–375, Amsterdam, The Netherlands, May 1990. IOS Press.
- [Wielinga *et al.*, 1993] B. J. Wielinga, A. Th. Schreiber, and J. A. Breuker. Modelling expertise. In A. Th. Schreiber, B. J. Wielinga, and J. A. Breuker, editors, *KADS: A Principled Approach to Knowledge-Based System Development*, pages 21–46. Academic Press, London, 1993.
- [Wols, 1993] R. Wols. Knowledge acquisition, modelling and formalisation for METEODES. Master's thesis, Dept. of Information Theory, Fac. of Electrical Engineering, Technical Univ. of Delft, July 1993. (in Dutch).