In: The Knowledge Engineering Review, 10(4), 1995.

Formal Specification Languages in Knowledge and Software Engineering

Dieter Fensel
University of Amsterdam, Department SWI, Roetersstraat 15, 1018 WB Amsterdam, the Netherlands.
E-mail: fensel@swi.psy.uva.nl

Abstract During the last years, a number of formal specification languages for knowledge-based systems (kbs) have been developed. Characteristics of such systems are a complex knowledge base and an inference engine which uses this knowledge to solve a given problem. Languages for kbs have to cover both these aspects. They have to provide a means to specify a complex and large amount of knowledge and they have to provide a means to specify the dynamic reasoning behavior of a kbs. Nevertheless, kbs are just a specific type of software system. Therefore it seems quite natural to compare formal languages for specifying kbs with formal languages which were developed by the software engineering community for specifying software systems. That is the subject of this paper.

Introduction

Over the last few years a number of *semiformal*, *formal*, and *executable* specification languages[1] have been developed for describing *knowledge-based systems* (*kbs*). These specification languages can be used to specify the knowledge which is required by the system as well as the reasoning process which uses this knowledge in order to solve the task which is assigned to the system. On the one hand, these languages should enable a specification which abstracts from implementation details. On the other hand, they should enable a detailed and precise specification of a kbs at a level which is beyond the scope of specifications in natural language. Surveys of these languages can be found in Treur & Wetter (1993) and Fensel & van Harmelen (1994). A short description of their history and usefulness is included in this issue (van Harmelen & Fensel, 1995).[2] The main purpose of this paper is not to provide a further survey of these languages. Instead, we compare this body of work with results achieved by the *software engineering* community. As kbs are a specific type of software artefact we should expect significant similarities between formal specification languages developed in knowledge engineering (KE) and in software engineering (SE). Therefore it is not an unreasonable idea to look at work which is done in the area of software engineering. The development of formal, executable, and semiformal specification languages has an almost twenty-five year tradition in SE and has led to a multitude of languages. In reaction to the software crisis from the late sixties a large number of formal specification languages were

1. An *informal specification technique* uses any natural language texts and arbitrary pictures. A *semi-formal specification technique* has a defined syntax, a number of predefined primitives, and allows the use of natural language in a limited manner. A *formal specification technique* has a semantics which is defined in a mathematical formalism. This semantics can be nonconstructive, that means it is not necessary that it is always computable. Still, it provides a precise formalism for specifying software and it can also be used to prove properties of such a specification. An *executable specification technique* has a constructive semantics which should be implemented to provide testing as a means of evaluating a specification.

2. See also ftp://swi.psy.uva.nl/pub/keml/keml.html at the World Wide Web.

developed which intend to improve the development process of software as well as the quality of its result. Most of these specification languages focus on a functional specification of the software artefact. That is, they specify what a software artefact must do without determining the way this functionality should be achieved, i.e. realized. A main concern of these languages is the development of correct software. Specifications define proof obligations for the specifier and later on for the programmer of the software. Program transformation calculi as in Bauer et al. (1987) or data reification and operation decomposition techniques as in Bicarregui et al. (1994) support the development of correct software from and in accordance with a formal specification.

A difference between these approaches and knowledge specification languages results from the fact that the latter do not aim at a pure functional specification. Specification languages from knowledge engineering also specify control over the use of the knowledge during the reasoning process. That is, they specify the way the functionality is achieved. In general, such differences are not surprising. As knowledge specification languages are restricted to a specific type of system we should expect specific features of these languages which distinguish them from "general purpose" languages in software engineering. The latter, after all, must cover a much broader collection of software artefacts. We will see that knowledge specification languages make much more use of conceptual models in describing a system. These models distinguish different types of knowledge and define object/meta-level relationships between them. The applicability of such conceptual models is gained by restricting the scope of the languages to a specific type of system. Specific aspects of kbs which result in specific requirements for specification languages are:[3]

(i) The *separation of knowledge and control*: The original production rule paradigm enabled the declarative specification of knowledge by a set of production rules. The dynamics of the reasoning process was provided by an inference engine using generic inference strategies as forward or backward chaining. Classical examples of this type of system are MYCIN (Clancey, 1987) and XCON (McDermott, 1982). A new level of defining control independently of the domain knowledge was reached by developing *task-specific shells* or so-called *role-limiting methods* (Chandrasekaran, 1986; Marcus, 1988). These shells fix a task-specific problem-solving strategy and distinguish the different types of domain knowledge required for it. Meanwhile, a collection of such task-specific *problem-solving methods* exists (see Breuker & van de Velde, 1994) which can be used to specify the reasoning process of a kbs independently from the domain knowledge.

(ii) The *generic specification of problem-solving methods*: The separation of knowledge and the control of the reasoning process using this knowledge enable the generic specification of this control knowledge. As a consequence two types of software reuse become possible: the problem-solving methods can be applied in different domains and the domain knowledge can be used by different methods.

(iii) *Object-meta relationship* between domain knowledge and inference processes: A problem-solving method defines and controls the use of domain knowledge for the inference process. It is therefore a meta-level description of the object-level knowledge. Most knowledge specification languages therefore provide a meta-level architecture to describe a kbs.

The discussion of the relationships between formal methods in KE and SE has three goals.

3. A fourth requirement will be given subsequently.

Firstly, it should provide a framework for applying general results achieved in SE for the specification of kbs. It should prevent the KE field from developing things a second time. Secondly, it should enable the possibility to generalize results achieved for specifying kbs to other types of systems which share similarities with kbs. Thirdly, the relationships between formal methods in both communities become even more important when integrated systems need to be specified. The integration of knowledge-based subcomponents into conventional software systems arises as a necessity when such components are applied in real-world environments.

Two limitations of our paper have to be mentioned. Firstly, we look at SE and KE as academic disciplines. That is, we discuss languages which are mentioned in the scientific literature. In the case of SE, most of the languages which we will discuss can report a number of applications. In the case of KE, there are as yet only very few applications of formal specification methods outside the academic world. Secondly, we focus our attention on specification languages. A different issue would be to compare methods or methodologies for the formal specification of systems. Methodological aspects like the combination of conceptual modelling techniques with formal specification techniques or formal support in deriving correct implementations from specifications are discussed only if they appear as features of these languages.

The contents of the paper is organized as follows. Section 1 introduces the main characteristics of specification languages for kbs. We have chosen the formal specification language (ML)[2] and the executable specification language KARL which both rely on the same semiformal specification technique—the KADS model of expertise—as a conceptual model to describe a kbs. These languages are contrasted with DESIRE which describes a kbs as a set of components (i.e., modules) with internal object/meta-level distinction and global control. Section 2 discusses specification languages originating from software engineering. From the overwhelming amount of work we have chosen traditional approaches from the property-oriented stream (i.e., functional specifications based on algebraic techniques) as well as from the model-oriented stream (i.e., VDM and Z). In addition, we discuss a more recent approach using algebraic specification techniques to describe a software system in an operational manner (i.e., Evolving Algebras). Structured analysis is used as an example of a semiformal specification technique in software engineering. Finally, in section 3 an overall comparison of these languages is made along criteria which are relevant in the context of specifying kbs.

1 Specification Languages in Knowledge Engineering

> The knowledge level "is characterized by knowledge as the medium and the principle of rationality as the law of behavior." (Newell, 1982)

At the knowledge level, the knowledge required by a kbs to solve its task in an efficient manner is described in an implementation-independent manner. It is described in terms of goals, operations, and knowledge about the relationships of goals and operations. At the symbol level, a specific computational agent is implemented which carries out the problem-solving process by means of a computer program. In terms of software engineering: a knowledge level description is a specification of the functionality of the desired system and the required knowledge. A symbol level description corresponds to a design specification or implementation. Distinguishing between the knowledge and the symbol levels therefore reflects the distinction of specification and design/implementation in software engineering. In software engineering, the distinction between a functional specification and the design/implementation of a system is often discussed as a separation of *what* and *how*.

> "The generation of system-level requirements is, to the extent possible, a pure *what*, addressing the desired characteristics of the complete system. The next steps, determining the next level of the hierarchy and allocating system requirements to the elements, are in fact a *how*." (Dorfman, 1990)

During the specification phase, *what* the system should do is established in interaction with the users. *How* the system has to do its tasks is defined during design and implementation (i.e., which algorithmic solution can be applied). This separation—even in the domain of software engineering this separation is often not practicable—does *not* work in the same way in the domain of knowledge-based systems, because a high amount of the problem-solving knowledge, i.e. knowledge about *how* to meet the requirements, is not a question of efficient algorithms and data structures, but exists as domain-specific and task-specific heuristics as a result of the experience of an expert. For many problems which are completely specifiable it is not possible to find an efficient algorithmic solution. Problems in diagnosis or design are easy to specify but it is not necessarily possible to derive an efficient algorithm from these specifications; domain-specific heuristics or domain-specific inference knowledge is needed for the efficient derivation of a solution. "In simple terms this means analysis is not simply interested in *what* happens, as in conventional systems, but also with *how* and *why"* (Brooking, 1986). One must not only acquire knowledge about what a solution for a given problem is, but also knowledge about how to derive such a solution in an efficient manner.

As a consequence the problem arises of distinguishing the two different kinds knowledge of *how* to solve a problem in knowledge engineering (KE). "There is a difference between what we would call respectively *knowledge-level control* and *symbol-level control*" (Schreiber, 1992). At the knowledge level there is a description of the domain knowledge and the problem-solving method which is required by an agent to solve the problem effectively and efficiently. This knowledge must already be modelled during the specification phase. At the symbol level there is a description of efficient algorithmic solutions and data structures for implementing an efficient computer program (i.e., a very specific agent). As in software engineering, this type of knowledge can be added during the design and implementation of the system. Therefore, a fourth requirement for languages specifying kbs is:

(iv) A language must *combine non-functional and functional specification techniques*: On the one hand, it must be possible to express algorithmic control over the execution of substeps. On the other hand, it must be possible to characterize substeps only functionally without making commitments to their algorithmic realization.

During the last years a number of formal or executable specification languages have been developed for describing a kbs at the "knowledge level". The majority of the specification languages for kbs are based on the *KADS model of expertise* (see Wielinga et al., 1992; Schreiber et al., 1993) or define their conceptual model as a modification of it. As a detailed discussion and comparison of KADS oriented languages has already be provided by Fensel & van Harmelen (1994), we choose only prototypical approaches of this group to illustrate the significant features of these languages. In fact, we have chosen the languages $(ML)^2$ (van

Harmelen & Balder, 1992), which was developed as part of the KADS projects (Wielinga et al., 1992; Schreiber et al., 1993 & 1994), and KARL (Angele et al., 1994; Fensel, 1995), which can be viewed as the executable part of $(ML)^2$. $(ML)^2$ is a *formalization* language for KADS models of expertise. Its expressive power (full first-order logic) is beyond the scope of computational functions. KARL is an *operational* language which restricts the expressive power by using a variant of Horn logic. By comparing both languages we can point out some significant differences when aiming at formal or operational specification languages. As the significant property of all of the KADS oriented languages is the use of the KADS model of expertise as a conceptual framework for specifying a system, we also have to introduce this model and its underlying philosophy.

The language DESIRE (van Langevelde et al., 1992 & 1993) is discussed as an example from languages which rely on a different conceptual model for describing a kbs: the notion of a compositional architecture. A kbs is decomposed into several interacting components. Each component contains a piece of knowledge at its object-layer and its own control defined at its internal meta-layer. The interaction between components is represented by transactions and the control flow between these modules is defined by a set of control rules. As will be shown later, the interaction with the user of a kbs can easily be integrated into its specification (by an additional module). In contrast, languages like $(ML)^2$ specify only the reasoning process of the kbs but not its interaction with the environment as this aspect is not regarded in the KADS model of expertise.

Common to all approaches is that a specification of a kbs has to cover three aspects: the specification of *static aspects* of a kbs, the specification of the *dynamic aspects* of a kbs (i.e., its reasoning), and the combination of both.

1.1 KADS Oriented Languages

In the following, we first introduce the KADS oriented point of view in knowledge engineering. Then, the formalization and operationalization languages $(ML)^2$ and KARL for KADS models of expertise are described. Finally, we briefly compare both languages.

1.1.1 Knowledge Level Modelling of Knowledge-Based Systems

A collection of models was developed in the CommonKADS project (Schreiber et al., 1994) to cover the different aspects of a kbs. The *organisation model* describes the main features of an organisation in which the kbs should be used. The *task model* describes the set of tasks which are performed by the organisation. The *agent model* describes the different agents which execute these tasks including the kbs, its environment, the user, etc. The specification of the interaction of the user with the kbs is given in the *communication model*. The *design model* describes the architecture of the implementation which realizes the kbs. The *model of expertise* is particularly significant for describing a kbs as it describes the different types of knowledge required by the kbs as well as the role of this knowledge in the reasoning process of the kbs. This last model distinguishes three different types of knowledge:

- The *domain layer* provides the domain-specific knowledge necessary for defining the task and for realizing the different inference steps of the problem-solving process.
- The *inference layer* defines the reasoning process of the kbs. It consists of inference actions, (static and dynamic) knowledge roles, and an inference structure. The *inference actions* define the elementary inference steps of the reasoning process. The *static roles*

define the role of the domain knowledge in the reasoning process. The *dynamic roles* are used to refer to intermediate results of the problem-solving process. The *inference structure* connects inference actions and roles and defines the dependencies between these elements.

- The *task layer* defines the goals of the reasoning process and the way to achieve the goals. It defines *control* over the execution of inference actions.

We use a small diagnostic example to illustrate the model of expertise. The domain layer provides causal knowledge which can be used to relate findings to diagnoses and knowledge which can be used to assign preferences to possible diagnoses. The task of the kbs consists of finding the diagnosis with the highest probability for a given set of symptoms. The inference layer consists of two inference actions:

- *generate*, creating possible hypotheses based on the given findings and the causal relationships at the domain layer.
- *select*, assigning a preference to hypotheses and selecting the diagnosis with the highest preference.

A simple control flow is defined by first executing the inference *generate* and then applying the inference *select* on its output. *generate* derives all possible hypotheses which could explain the findings and *select* chooses the hypothesis with the highest preference (i.e., highest probability). The complete model is given in Figure 1. It is clear that such an exhaustive search could not normally be done for realistic domains. But we tried to keep the example as simple as possible and we will use it during the entire paper to illustrate the

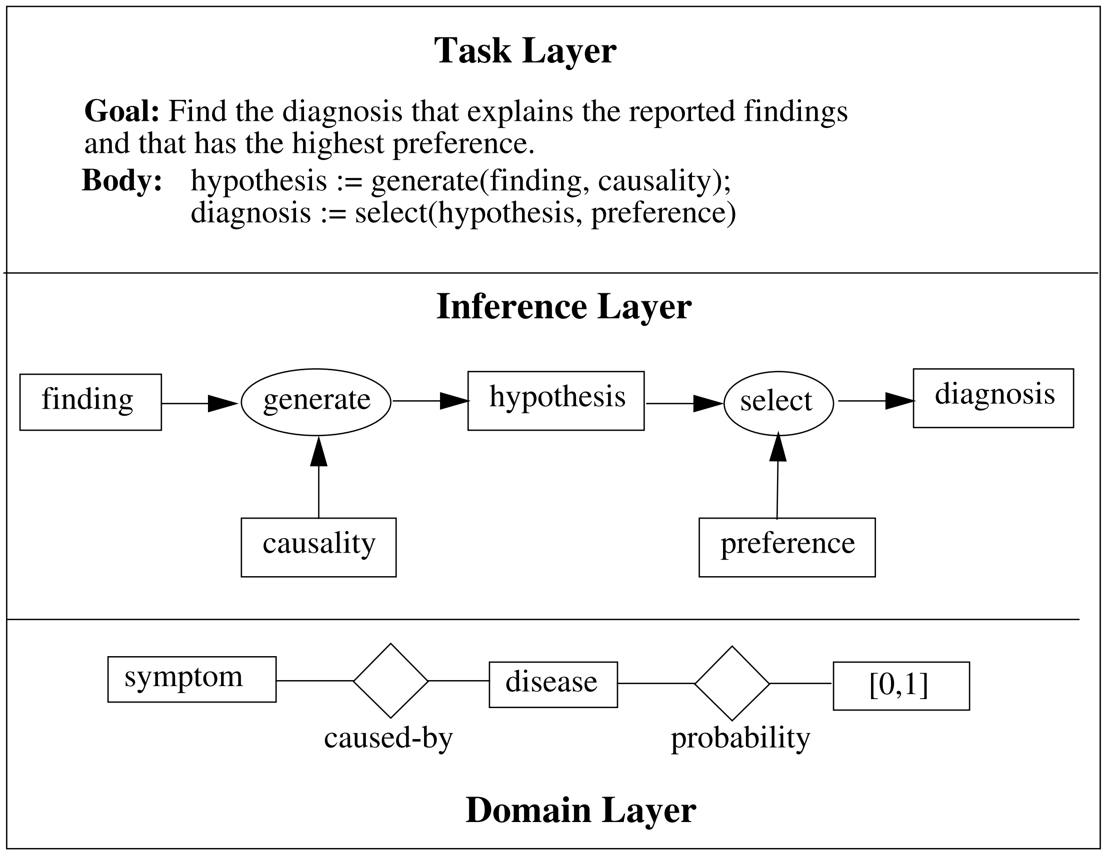


Fig. 1 A model of expertise for a simplified diagnostic task.

different specification approaches.[4]

The main point of a model of expertise is the separation of domain knowledge and control knowledge. The domain layer contains the static knowledge from the application domain and its terminology. The inference and task layers describe the dynamic reasoning process of the system. The inference layer defines the elementary inference steps, the relations between them, and the role of the domain knowledge for the reasoning process. In our example, the causal relationship is used by the *generate* inference step and the knowledge about probabilities is used by the *select* step. The description at the task layer completes the definition of the dynamics by defining control over the execution of the inference steps. The inference layer can be viewed as defining the vocabulary to express the control at the task layer. The distinction between the domain-specific knowledge and the domain-independent description of the reasoning process enables the *reuse of domain knowledge* for different task and reasoning strategies and the *reuse of reasoning strategies* (called *problem-solving methods*) in different domains. For instance, the problem-solving method in Figure 1 (i.e., the inference and task layers) can be applied in medical or technical application areas. In contrast to general-purpose methods like generate-and-test or search strategies like hill climbing, beam search, or chronological backtracking, such a problem-solving method is restricted to a specific type of problems (i.e., to a specific task). A library of reusable problem-solving methods is provided by Breuker & van de Velde (1994). Large-scale reuse of domain knowledge is studied in the KAKTUS project (cf. Wielinga & Schreiber, 1994; Schreiber et al., 1995).

Without using formal specification languages, the semantics of the elementary elements of a model of expertise have to be defined by using natural language. KARL and $(ML)^2$ have been developed to formalize these elementary elements.

1.1.2 $(ML)^2$

The language $(ML)^2$ (van Harmelen & Balder, 1992) provides a formal specification language for the KADS model of expertise by combining three types of logic: order-sorted first-order logic extended by modularization for specifying the domain layer, first-order meta-logic for specifying the inference layer, and quantified dynamic logic (Harel, 1984), which was originally developed for the verification of procedural programs, for specifying the task layer. In the following, we discuss domain, inference, and task layers of a model in $(ML)^2$.

Domain layer. The sublanguage of $(ML)^2$ used to model a domain layer is order-sorted first-order logic extended by modularisation. Instances are modelled by constants, and sorts can be used to model classes of such constants. Sorts can be arranged in an is-a hierarchy. Relationships between concepts are modelled by predicates of the according sorts. Attributes of concepts are modelled by functions. Arbitrary first-order theories can be used to specify

4. A further simplification is that we regard only *a priori* probabilities for the diagnoses. Therefore, we do not have to specify any probabilistic inferences. The definition of an appropriate calculus for probabilities is not at all an implementational detail which could be skipped during knowledge acquisition. Each of these existing calculi make specific assumptions about the domain and the task of the kbs. The specification of the appropriate calculi is therefore part of the knowledge acquisition process. None of the knowledge specification languages provides a built-in calculi for probabilities as there exists no golden standard for them. However, each of the languages provides language primitives which enable the specification of an inference calculus for probabilities according to a given domain and task.

theory *domain layer*
signature
sorts *symptom*, *disease*, *real*;
constants
no-fever, *low-fever*, *high-fever* : *symptom*;
healthy, *influenza*, *pneumonia* : *disease*;
0.5,0.1,0.05 : *real*;
functions *probability* : *disease* $\rightarrow$ *real*;
predicates
= : (*real*,*real*); > : (*real*,*real*);
caused-by : (*symptom*,*disease*);
prefer : (*disease*,*disease*);
actual-symptoms : *symptom*;
variables X_1, X_2 : *disease*; Y_1, Y_2 : *real*;
axioms
actual-symptoms(*low-fever*);
caused-by(*no-fever*,*healthy*);
caused-by(*low-fever*,*influenza*);
caused-by(*low-fever*,*pneumonia*);
caused-by(*high-fever*,*pneumonia*);
=(*probability*(*healthy*),0.5);
=(*probability*(*influenza*),0.1);
=(*probability*(*pneumonia*),0.05);
$prefer(X_1,X_2) \leftarrow =(probability(X_1),Y_1) \wedge =(probability(X_2),Y_2) \wedge Y_1 > Y_2$;
endtheory

Fig. 2 A domain layer in $(ML)^2$.

relationships. The specification of a domain layer can be divided into several modules. Such a module or theory defines a signature (i.e., sorts, constants, functions, and predicates) and defines axioms (i.e., logical formulae). These modules, i.e. subtheories, can be combined by a union operator. In the following, we define the domain layer for our running example. For simplicity, we define the entire domain layer in one module; see Figure 2.

Inference layer. In $(ML)^2$ every inference action and every knowledge role is described by a theory similar to domain layer theories. An inference action (called **p**rimitive **i**nference **a**ction in $(ML)^2$) is described by a predicate and a logical theory. The inference actions *generate* and *select* are modelled by two predicates:

$\text{pia}_{generate}(finding(X), causality(finding(X),hypothesis(Y)), hypothesis(Y))$
$\text{pia}_{select}(hypothesis(X), preference(Z), diagnosis(Y))$

The description of the inference action *select* is given in Figure 3. $(ML)^2$ also uses order-sorted logic at this layer, but we leave out the sorts in Figure 3 for brevity.

The inference layer is modelled as a meta-language of the domain layer. This meta-relation allows the inference-layer to specify properties of relations over domain-layer formulae without resorting to second-order logic. Object- and meta-language are connected by a naming relation and reflection rules. At the inference layer a lift operation is defined for every knowledge role which is connected to the domain layer. The lift operator defines a naming relation by mapping expressions of the domain layer to variable-free terms at the inference layer. This lift-operator is defined as a system of rewrite rules that translate domain-layer sentences into inference-layer terms.

theory *select*
  input roles *hypothesis*, *preference*;
  output roles *diagnosis*;
  signature
    predicates pia_{select};
    variables X, Y, Z;
  axioms
    $\mathrm{pia}_{select}(hypothesis(X), preference(Z), diagnosis(X)) \leftarrow$
      $\mathrm{input}_{hypothesis}(hypothesis(X)) \wedge$
      $\mathrm{input}_{preference}(preference(Z)) \wedge$
      $\neg(\exists Y : \mathrm{input}_{hypothesis}(hypothesis(Y)) \wedge pref(hypothesis(Y),hypothesis(X)) \in Z)$
endtheory

Fig. 3 Inference actions in $(\mathrm{ML})^2$.

In Figure 4, we introduce the lift definitions for the knowledge roles *preference*, *hypothesis* and *diagnosis*.[5] The knowledge role *preference* is connected to the domain layer as it provides domain knowledge for the reasoning process. On the other hand, the knowledge role *hypothesis* collects intermediate results from the problem-solving process and provides it for another inference action. As it obtains its contents from an inference action it does not require mapping definitions which would connect it to the domain layer. The knowledge role *diagnosis* collects the results of the problem-solving process and has no connection to the domain layer either.

The input predicates used in the definition of the inference action $\mathrm{pia}_{generate}$ and pia_{select} have not yet been defined. For this purpose, reflection rules are provided which connect truth in object- and meta-logic.

$\mathrm{input}_{finding}(finding(X)) =_{\mathrm{def}} \mathrm{ask}_{\in}(domain\ layer, finding(X))$
$\mathrm{input}_{causality}(causality(X,Y)) =_{\mathrm{def}} \mathrm{ask}_{\in}(domain\ layer, causality(X,Y))$

lift-definition *preference*
  from *domain layer*;
  to *select*;
  signature
    sorts *boolean*; *quoted-term*
    constants "X", "Y";
    functions *preference*: (*quoted-term* , *quoted-term*) $\rightarrow$ *boolean*;
  mapping
    lift(*domain layer*,*prefer*(X,Y)) $\rightarrow$ *pref*("X", "Y");
end lift-definition

lift-definition *hypothesis*
  to *generate*, *select*;
  signature
    sorts *boolean*; *term;*
    functions *hypothesis*: *term* $\rightarrow$ *boolean*;
end lift-definition

lift-definition *diagnosis*
  to *select*;
  signature
    sorts *boolean*; *term;*
    functions *diagnosis*: *term* $\rightarrow$ *boolean*;
end lift-definition

Fig. 4 Connecting domain and inference layers in $(\mathrm{ML})^2$.

5. For the sake of limited space we skip the lift definitions of *finding* and *causality.*

$\text{input}_{preference}(Z) =_{\text{def}} \{ \; pref(hypothesis(X),hypothesis(Y)) \mid \text{ask}_{\vdash}(domain\ layer,prefer(X,Y)) \}$

$\text{ask}_{\in}(theory,X)$ is true iff X is an axiom of *theory* and $\text{ask}_{\vdash}(theory,X)$ is true iff X is a logical consequence of *theory*. The knowledge role *hypothesis* does not provide domain knowledge for the inference actions. It collects the output of the inference action *generate* and provides it as an input to the inference action *select*. This dynamic character of *hypothesis* makes it necessary to define the input predicate $\text{input}_{hypothesis}$ at the task layer. The knowledge role *diagnosis* is used as an output role only and therefore requires no input predicate.

Task layer. Quantified-dynamic logic is used to specify dynamic control at the task layer. Every predicate specifying an inference action at the inference layer together with the test operator *?* is regarded as an elementary program statement and the knowledge roles are used as input and output parameters of such programs. For every such elementary program a history variable V_{pia_i} is defined which stores the input-output pairs for every execution step.

The key idea is to non-deterministically choose a value binding of a logical variable by the test operator and store this value in a state variable.

Four types of task-layer operations are available for each inference action pia_i: checking whether an instantiation exists, checking whether an instantiation has already been computed, checking whether more instantiations exist, and actually computing and storing a new instantiation:

$has\text{-}solution\text{-}pia_i\,(I,O) =_{\text{def}} \text{pia}_\text{i}(I,O)$

$old\text{-}solution\text{-}pia_i\,(I,O) =_{\text{def}} ((I,O) \in V_{pia_i})$

$more\text{-}solution\text{-}pia_i(I,O) =_{\text{def}} (has\text{-}solution\text{-}pia_i(I,O) \wedge \neg old\text{-}solution\text{-}pia_i(I,O))$

The important program is $give\text{-}solution\text{-}pia_i$ which gives one possible solution:

$give\text{-}solution\text{-}pia_i\,(I,O) =_{\text{def}} (more\text{-}solution\text{-}pia_i(I,O)?;\ V_{pia_i} := <(I,O) \mid V_{pia_i}>$

Note that $old\text{-}solution\text{-}pia_i$ versus V_{pia_i} is an administration for the non-deterministic execution of $give\text{-}solution\text{-}pia_i$ necessary to ensure the derivation of new instantiations of the predicate.

These primitive programs and predicates can be combined using sequential composition, non-deterministic iteration and non-deterministic choice. These combinations are rich enough to model more standard constructions like deterministic iteration and conditional statements.

For our example, we have to define the input predicate $\text{input}_{hypothesis}$ and the control flow between the inference actions. The knowledge role *hypothesis* collects the output of the inference action *generate* and provides it as input to the inference action *select*. The following definition of the input predicate is the way in which $(ML)^2$ can be used to define dataflow between inferences.

$\text{input}_{hypothesis}(X) =_{\text{def}} \exists\, I_1,I_2 \text{ with } (I_1,I_2,X) \in V_{pia_{generate}}$

The task layer of our example is given in Figure 5.

while *more-solution-pia*$_{generate}$(*finding*(*X*),*causality*(*finding*(*X*),*hypothesis*(*Y*)),*hypothesis*(*Y*))
do *give-solution-pia*$_{generate}$(*finding*(*X*),*causality*(*finding*(*X*),*hypothesis*(*Y*)),*hypothesis*(*Y*)) **enddo**
give-solution-pia$_{select}$(*hypothesis*(*X*),*preference*(*Z*),*diagnosis*(*X*))

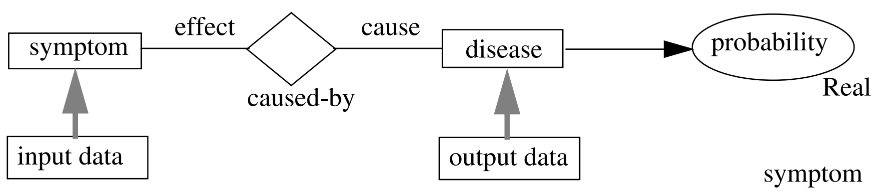
Fig. 5 A task layer in $(ML)^2$.

1.1.3 KARL

The language KARL (Angele et al., 1994; Fensel, 1995) was developed as part of the MIKE project (Angele et al., 1993) and provides a formal and executable specification language for the KADS model of expertise by combining two types of logic: Logical-KARL (L-KARL) and Procedural-KARL (P-KARL). L-KARL, a variant of Frame Logic (Kifer et al., 1993), is provided to specify domain and inference layers. It combines first-order logic with semantic data modelling primitives (see Brodie (1984) for an introduction to semantic data models). A restricted version of dynamic logic is provided by P-KARL to specify a task layer. Executability is achieved by restricting Frame logic to Horn logic with stratified negation (Przymusinski, 1988) and by restricting dynamic logic to regular and deterministic programs. Again, we will discuss the domain, inference, and task layers in KARL.

Domain layer. KARL uses L-KARL to describe the domain layer. It provides predicates, classes, class hierarchies, single- and set-valued attributes with domain and range restrictions, and multiple attribute inheritance for modelling terminological domain knowledge. The derivation of new object denotations can be expressed by functions. The domain layer of our running example is given in Figure 6. As it is a simple example the domain layer contains only ground facts and some terminological definitions given by the graphical language of KARL.

Inference layer. The same language L-KARL as is used at the domain layer is provided for specifying inference actions and roles at the inference layer. KARL distinguishes three types of knowledge roles. Views define an upward translation from the domain layer to the inference layer (giving read-access). Terminators define a downward translation from the inference layer to the domain layer (giving write-access). Stores provide the input or output



symptom

E-ID term
no-fever
low-fever
high-fever

disease

E-ID term	probability
healthy	0.5
influenza	0.1
pneumonia	0.05

caused-by

effect	cause
no-fever	healthy
low-fever	influenza
low-fever	pneumonia
high-fever	pneumonia

input data

E-ID term
low-fever

Fig. 6 A domain layer in KARL.

of inference actions. Whereas views and terminators are used to link a domain layer with a generic inference layer, stores are used to model the dataflow dependencies between inference actions. The definitions of the inference actions, stores, views, and terminators of our example are given in Figure 7.

Task layer. KARL uses the logical language Procedural-KARL (KARL), a variant of dynamic logic, at the task-layer. Therefore, it can be used in a similar way as procedural programming languages to express control. The primitive programs correspond to *calling an inference action*, and atomic formulae indicate whether knowledge roles contain elements of a given class. Such primitive programs and atomic formulae can be arranged into sequences, loops, and alternatives. Programs may be combined to named subtasks, similar to procedures in programming languages. The task layer of our example looks like:

hypothesis := *generate*(*finding*); *diagnosis* := *select*(*hypothesis*)

Each inference action defines a function symbol used in assignments. Each store and terminator is modelled by a (program) variable. The value assignments of the variables are used to represent the current state of the reasoning process.

1.1.4 Differences between KARL and $(ML)^2$

Two remarks can be made when comparing the **conceptual models** of $(ML)^2$ and KARL: on the one hand, KARL provides much stronger support in modelling terminological knowledge at the domain and inference layers. The KARL model distinguishes values, objects, classes, attributes with domain and range restrictions, predicates, is-a relationships between classes, etc. KARL integrates semantic data modelling primitives into a logical framework whereas $(ML)^2$ provides only the language primitives of first-order logic (i.e., sorts, predicates and constants) as modelling primitives. On the other hand, KARL does not provide an object-meta-logic relationship between the logical languages used to describe domain and inference layers and restricts the logical language to Horn logic (with restricted negation in the body of Horn clauses).

From a **semantical point of view**, two main differences arise: KARL uses the minimal model semantics of logic programming for the domain and inference layers, whereas $(ML)^2$ relies on the standard model-theoretical semantics of predicate logic. At the task layer, inference actions are modelled as predicates in $(ML)^2$ and as functions in KARL. In KARL, each inference action defines a function which is used to interpret a function symbol used in an assignment in dynamic logic. The execution of an inference action delivers one instantiation of the predicate in $(ML)^2$ and all "instantiations" in KARL as the complete minimal Herbrand model of the logical theory describing the inference action and the facts of the input roles is evaluated. Therefore, there is no non-deterministic choice of one instantiation of the inference actions and also no history variable in KARL which is necessary in $(ML)^2$ to prevent the rederivation of "old" values.

"A specification must be operational" (Balzer & Goldman, 1979): KARL is an **executable specification language** as the logical language at the domain and inference layers is restricted to Horn logic (extended by stratified negation) and the dynamic logic at the task layer is restricted to regular and deterministic programs (cf. Kozen, 1990). For $(ML)^2$, the evaluation of a specification is possible for the domain and inference layers with theorem-proving techniques but no support is provided for the overall evaluation including the specification of the dynamic aspects of the reasoning process at the task layer. Hayes & Jones

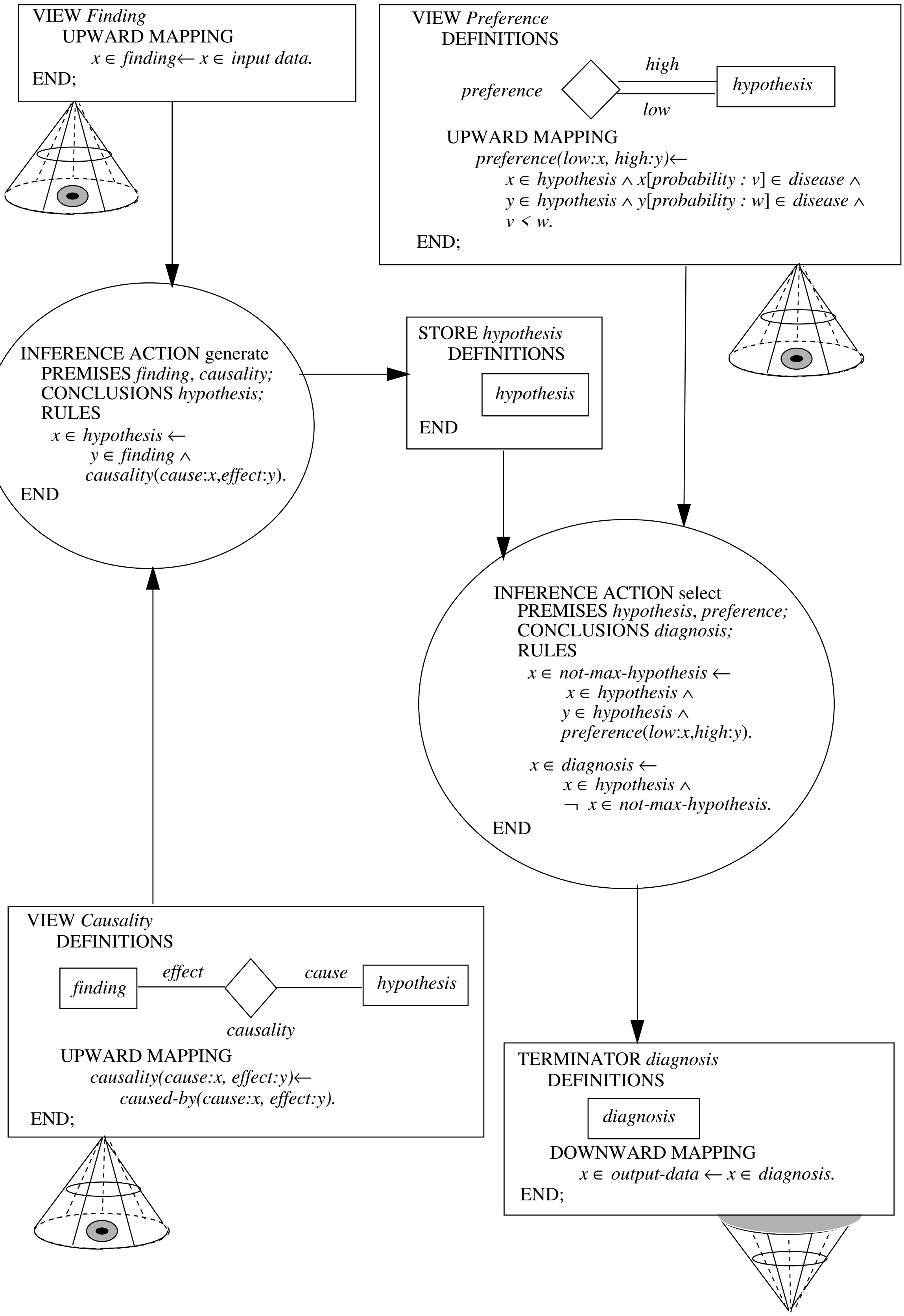


Fig. 7 An inference layer in KARL.

(1989) and Fuchs (1992) carried out a debate as to whether specification languages should be operational or not. In the domains of software engineering, information system development, and knowledge engineering, operational as well as formal specification languages can be found. This seems therefore to be an open debate. Two major objections to operational specification languages are:

- The executability restricts the expressive power especially if efficient execution is required to support prototyping in a meaningful manner.
- Executable specifications can unnecessarily constrain the choice of possible implementations.

Both of these objections have been experienced during the design of KARL. The logical language is restricted to Horn logic and the specification of control requires a deterministic (over-)specification. On the other hand, testing is a powerful tool for evaluating a specification and neither symbolic execution nor partial verification can deliver the same support for it. In general, the use of techniques from logic programming, which integrate a declarative semantics with an operational one, make the distinction between the two types of specification approaches less sharp.

1.2 DESIRE

DESIRE (van Langevelde et al., 1992; van Langevelde et al., 1993)[6] stands for *DEsign and Specification of Interacting REasoning components*. Three main principles underlie the specification of a kbs with DESIRE:

- DESIRE distinguishes a local and a global view in describing a system. A task is decomposed into several components. Each component defines a local view of the knowledge. The global view is defined by decomposing the whole system into modules and by introducing interactions between the modules.
- Another distinction is made between static and dynamic aspects. The specification of the static aspects covers the data and the knowledge of the system. The specification of its dynamic aspects covers the dynamic reasoning behavior of the system. DESIRE includes the specification of control knowledge which guides the reasoning process of the system.
- The third distinction concerns the difference of object-level and meta-level reasoning. At the object-level, the system reasons about the world state. Knowledge about how to use this knowledge to guide the reasoning process is specified at the meta-level. The meta-level reasons about controlling the use of the knowledge specified at the object-level during the reasoning process. The meta-level describes the dynamic aspects of the object-level in a declarative fashion.

Important distinctions between DESIRE and the specification languages which were discussed above are as follows. First, DESIRE does not rely on the KADS model of expertise as a conceptual framework for specifying the system. Second, DESIRE does not rely on the distinction of six different model types as is in KADS. Actually, a specification in DESIRE could also cover the user/system-interaction (Brazier & Treur, 1994), which is described in the model of communication in KADS. The modularization and interaction concept of DESIRE can be naturally extended to specify multi-agent systems as described by Brazier et al. (1995a). Third, DESIRE uses its object/meta-level distinction *to specify and to reason*

6. A good introduction is provided by Geelen et al. (1991).

about flexible control of object-level inferences whereas languages such as (ML)[2] or KARL define control of object-level inferences by a procedural language. DESIRE also provides a much more detailed means to express control of the object-level inferences.

1.2.1 The Global View

The definition of the global level includes the *decomposition* of the entire functionality into components (i.e., modules), the definition of the *dataflows* between the modules by means of transformations, and the global *control flow*. Recently, *hierarchical structuring primitives* were introduced by Brazier et al. (1995b) which enable hierarchical specification of the decomposition reflecting the hierarchical refinement from tasks into subtasks. Modules, which correspond to elementary subtasks, and their data and control flows can be grouped together to form larger components. For reasons of simplicity, we will not discuss this additional structuring primitive in the paper and refer the reader to Brazier et al. (1995b).

The entire task of a kbs is decomposed into subtasks and each module achieves a subtask of the complete functionality. Each module has its own *information state* which can change as the result of an interaction with another module or as a result of its own activation which may provide data for another module. DESIRE distinguishes conventional and reasoning modules. Conventional modules can provide information from the environment (databases, user, etc.) whereas reasoning modules describe how some output facts can be derived from input facts by means of a knowledge base. Each reasoning module contains object and meta-level information. The meta-level can be used to express control over the reasoning process at the object level.

The dataflow between modules is represented by transformations between modules. DESIRE distinguishes four main *transformation* types (cf. Geelen et al., 1991; Treur, 1992) as pictured in Figure 8:

- *Object interaction*: Object-level information from module M_1 is used as object-level input for module M_2.
- *Meta interaction*: Meta-level information from module M_1 is used as meta-level input for module M_2.
- *Upward reflection*: Meta-level information from module M_2 is used as object-level input for module M_3. Meta-level facts of module M_2 are used in module M_3 as object-level facts. Therefore, module M_3 reasons about module M_2.
- *Downward reflection*: Object-level information from module M_3 is used as meta-level input for module M_2. Object-level conclusions of module M_3 are used as meta-level facts for module M_2. Therefore, the results of the reasoning process of M_3 are used to control the reasoning process of module M_2.

By using transformations object-meta distinctions between components are made explicit, allowing an unlimited number of meta-levels to describe the entire system.

The global control flow of the system is specified by supervisor rules which determine the starting point of the reasoning process (i.e., the module which starts the reasoning process) and the possible transitions. The basic primitives of the control language are the activation of a module and preconditions for its activation. The general outline of a control rule is given in Figure 9. The condition that specifies when a module N has to be activated is described in terms of the termination status of some other module M.[7] If module M has achieved its goal, specified by the *target set*$_1$, its termination status is **succeeded**, otherwise it is **failed**. A third

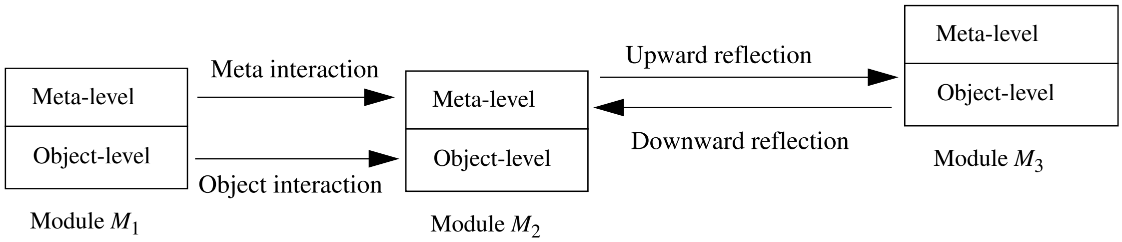


Fig. 8 The four transformation types in DESIRE.

IF termination(M, *target set*$_1$, **succeeded**)
    THEN next-module(N, *target set*$_2$*, request type, exhaustiveness*)
    AND next-pre-trans(T_1,...,T_n)

Fig. 9 The general outline of a control rule in DESIRE.

possibility is that a module gets **suspended** when its reasoning process stops with a request for additional input facts required to achieve its reasoning goal. The *target set*$_2$ specifies the sort of literals which should be derived by the newly activated module N (i.e., it specifies the goal of the local reasoning process). The two different request types **data-driven** and **goal-driven** are provided. In the former case, the module derives target literals as far as possible given the available input, and will not generate requests for additional information. In the latter case, the module poses a request for further input literals if it needs an unknown input literal during the derivation process of a target literal. DESIRE distinguishes different degrees of exhaustiveness with respect to the target set. For example: derive the truth value of *one* literal of the target set (**any**); derive the truth value of *one* literal of the target set which has *not* been derived before (**any-new**); derive the truth values of all literals of the target set for which it is possible (**all-possible**); or derive the truth value of *all* literals of the target set (**every**). Before executing the module N, the transformations T_1,...,T_n are executed providing the module N with new input.

1.2.2 The Local View

At the local level, the modules and the transformations have to be specified in more detail. The specification schema of a reasoning *module* is given in Figure 10. The signatures of a module are divided into the input signature, the output signature, and the internal (hidden) signature of the module. The input and output signatures can be used in the specification of transformations. The target set specifies the set of atoms one is interested in as result of the

Module <name> : **reasoning**
    input <signature>
    output <signature>
    target-sets <list-of-target-set-names>
    initial meta-facts
        <initial-meta-facts-specification>
    internal <signature>
    knowledge base
        <rules>
endmod

Fig. 10 The specification of a reasoning module in DESIRE.

7. Or a list of modules if appropriate.

reasoning process of the module. For each target atom, it is further specified whether it should be confirmed (i.e., found to be true), rejected (i.e., found to be false), or just determined (i.e, found to be true or false).

Each module is divided into an *object level* (the knowledge base) and a *meta level*. The language for specifying an object level is many-sorted predicate logic with three truth values *true*, *false*, and *unknown*. The signature (input + output + internal signature) defines sorts, constants, function symbols, and predicate symbols. An assignment of a truth value to all ground atoms of the language is called an *information state*. During the reasoning process of the module this information state is changed.

The signature of the meta-level language is implicitly given by the object-level language and some standard predicates which are used to express meta-information over the information state of the object level. The meta-level facts are divided into *targets* and *assumptions* of the reasoning process at the object level (meta-input facts), and *epistemic facts* and *requests* from the reasoning process at the object level (meta-output facts). A *target* specifies an object-level output fact a component should derive and an *assumption* specifies an object-level input fact that is being assumed by the component. An *epistemic* fact describes the truth value of an object-level fact and a *request* specifies an object-level input fact which must be known by the module to complete the object-level reasoning process. The initial control status of a module can be specified by the initial meta-level facts.

At both levels, formulas are all-quantified disjunctions of literals which are written as implications (**if** literal-conjunction **then** literal). Existential quantification is not provided. As the conclusion of such an implication could be a negated literal the language is not restricted to Horn logic.[8]

The logic at the meta level is two-valued. That is, the three-valued logic at the object level has to be mapped on the two-valued logic at the meta level. Table 1 shows this mapping for the three standard meta predicates *unknown*(*X*), *false*(*X*), and *true*(*X*).

Table 1. Truth values of meta level predicates in DESIRE

object-fact	*unknown*(*object-fact*)	*false*(*object-fact*)	*true*(*object-fact*)
unknown	true	false	false
false	false	true	false
true	false	false	true

A *transformation* has to define the correspondence of the output atoms of one module with the input atoms of another module. In addition, it is necessary to describe how the truth values of the atoms are related. The specification of the **domain** and **co-domain** of a transformation are part of the global view and define the connection between modules. The *from-type* can be **object**, **epistemic**, or **request**, the *to-type* can be **object**, **assumption**, or **target**. In total, nine different variants of transformations can be defined (see Treur, 1992). Each of them fall into one of the four types of transformation which were mentioned earlier.

1.2.3 The Example in DESIRE

In the following, we use a part of our running diagnosis example. In fact, we refine the hypothesis generation step following an example given by van Langevelde et al. (1992). The

8. The authors of DESIRE make no precise judgement about the expressive power of their sublanguage of first-order logic.

user input of findings (symptoms) is viewed to be given to the system in the $(ML)^2$ and KARL models. The hypothesis generation step derives all possible hypothesis for the given findings. In the following, we refine this inference action *generate* into three components, as DESIRE also aims to represent the user/system interaction and the detailed control of the object-level inference process. We specify a component, *symptom interpretation,* which reasons about the given symptoms and possible hypotheses to explain them. It also specifies the interaction with the user if the truth values of further symptoms are required by the reasoning process. A component *hypothesis generation* derives possible hypotheses controlled by the meta-level of this component. A third component *world* represents the user.

Figure 11 defines the global control and dataflow of our example. Three modules and four transformations are defined. The module *hypothesis generation* generates possible hypotheses. The module *symptom interpretation* checks whether a possible hypothesis is confirmed by the available symptoms. If it needs further symptoms it asks the module *world* for further input. Control rule (1) starts the entire system by executing the module *hypothesis generation*. If this module terminates, either control rule (6) or (2) applies. If the module *symptom interpretation* is activated by control rule (2) it can either **succeed**, **fail** or it can get **suspended**. In the first case control rule (7) is applied and a hypothesis which is consistent with the given symptoms is found. In the second case control rule (3) applies which leads again to the activation of module *hypothesis generation*. In the third case, the module world is activated by rule (4) and if it succeeds it leads to the reactivation of module *symptom*

(1) **IF start**
THEN next-module(*hypothesis generation,initial targets*,**data-driven**,**any**)
AND next-pre-trans(*reflect up*)
(2) **IF termination**(*hypothesis generation,initial targets*,**succeeded**)
THEN next-module(*symptom interpretation*, *dynamic targets*,**goal-driven**,**every**)
AND next-pre-trans(*reflect down*)
(3) **IF termination**(*symptom interpretation*, *dynamic targets*,**failed**)
THEN next-module(*hypothesis generation,initial targets*,**data-driven**,**any-new**)
AND next-pre-trans(*reflect up*)
(4) **IF suspended**(*symptom interpretation*)
THEN next-module(*world*, *dynamic targets*,**goal-driven**,**any**)
AND next-pre-trans(*ask*)
(5) **IF termination**(*world*, *dynamic targets*, **succeeded**)
THEN next-to-resume(*symptom interpretation*)
AND next-pre-trans(*answer*)
(6) **IF termination**(*hypothesis generation,initial targets*,**failed**) **THEN stop**
(7) **IF termination**(*symptom interpretation*, *dynamic targets*,**succeeded**) **THEN stop**

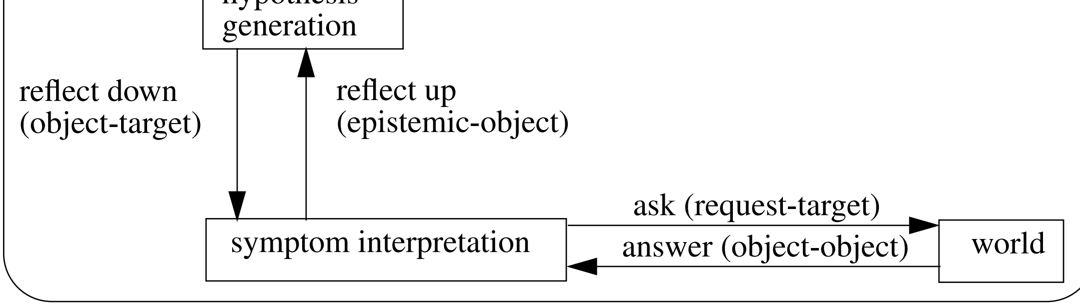


Fig. 11 The global level in DESIRE

signature *symptoms*
  sorts *fever-degree*
  objects
    no,low, high : *fever-degree*;
  relations
    fever : *fever-degree*;
endsig
signature *hypotheses*
  relations *healthy, influenza,pneumonia*;
endsig

signature *meta-hyp*
  sorts
    hypotheses
  objects
    healthy, influenza,pneumonia: *hypotheses*;
endsig

Fig. 12 Signature definitions.

interpretation by rule (5).

Four transformations are defined. The transformation *reflect down* provides facts from the object-level of *hypothesis generation* as **target** facts of the meta-level of *symptom interpretation*. A possible hypothesis which is generated by the module *hypothesis generation* is passed by this transformation to the module *symptom interpretation* to check whether it coincides with the available symptoms. The transformation *reflect up* provides **epistemic** facts from the meta-level of *symptom interpretation* as object-level facts of *hypothesis generation*. A meta-level fact from *symptom interpretation* expressing the truth value of object-level expressions becomes an atom at the object-level of *hypothesis generation*. This input is used by *hypothesis generation* to derive a new possible hypothesis. The transformation *ask* is a meta-level interaction where a **request** of *symptom interpretation* is passed as a **target** to the module *world*. This transformation covers the case where the module *symptom interpretation* needs the truth value of a new symptom in order to evaluate a possible hypothesis. Finally, the transformation *answer* passes object-level facts from *world* to *symptom interpretation*. Here, the request of *symptom interpretation* for the truth value of a new symptom is fulfilled.

After defining the global level, the three modules and their transformations have to be specified. First, we define in Figure 12 signatures which are used in the modules. Figure 13 provides the definitions of the reasoning modules *symptom interpretation* and *hypothesis generation*. The strategy of *hypothesis generation* is to use the hypothesis *healthy* as a default hypothesis. If *symptom interpretation* returns that this hypothesis is wrong according to its knowledge base and the requested truth values of *symptoms*, one the other possible hypotheses *influenza* and *pneumonia* is generated. The dataflow transformations *reflect down, reflect up, ask,* and *answer* remain to be defined. Figure 14 provides the definition of *reflect down* and *reflect up*. The first transition passes a reasoning target (*possible hypothesis*) which should be confirmed (other possibilities would be **reject** or **confirm**) from *hypothesis generation* to *symptom interpretation*. This downward reflection has to map a three-valued logic onto a two-valued one. The second transformation returns the truth value of the provided hypothesis.

1.2.4 Linear Partial Temporal Logic as Semantics

In this section we give a rough survey on the semantics of DESIRE. For a general outline see Treur (1994) and for details Gavrila & Treur (1994). DESIRE uses *partial models* to represent the *information states* of the object-level of a module. One information state is represented by one partial model: ground atoms can have the truth value true, false or

module *hypothesis generation* : **reasoning**
input signature *input-hg*
signature *meta-hyp*;
relations
si-false : *hypotheses*;
endsig
output signature *output-hg*
signature *meta-hyp*;
relations
possible-hypothesis : *hypotheses*;
endsig
target sets *initial targets*
initial meta-facts
target(*initial targets,*
possible-hypothesis
(*X*:*hypotheses*),***determine***);
knowledge base
if not *si-false*(*healthy*)
then *possible-hypothesis* (*healthy*);
if *si-false*(*healthy*)
then *possible-hypothesis* (*influenza*);
if *si-false*(*healthy*)
then *possible-hypothesis* (*pneumonia*);
endmod

module *symptom interpretation* : **reasoning**
input signature *symptoms*;
output signature *output-si*
signature *meta-hyp*;
endsig
target sets *dynamic targets*
initial meta-facts
requestable(*fever*(*D*:*fever-degree*));
knowledge base
if *fever*(*no*) then *healthy*;
if *fever*(*no*) then not *pneumonia*;
if *fever*(*no*) then not *influenza*;
if *fever*(*low*) then *influenza*;
if *fever*(*low*) then not *healthy*;
if *fever*(*low*) then not *pneumonia*;
if *fever*(*high*) then *influenza*;
if *fever*(*high*) then *pneumonia*;
if *fever*(*high*) then not *healthy*;
endmod

Fig. 13 The modules *hypothesis generation* and *symptom interpretation*.

unknown. During conservative reasoning, unknown facts can change to true or false. This three-valued logic is used only for the object-level. The truth values of the meta-level of a module are always complete, see table 1. The reasoning process at the meta-layer is non-conservative. For example, the derivation of a truth value for an object-level fact which was originally unknown changes the truth value of *unknown*(*object-level fact*) from *true* to *false*. The information state of an entire module is therefore a pair consisting of a partial model for the object-level and a complete model for the meta-level.

Linear temporal logic with partial models as states is used to specify the reasoning trace of a module (and also of the whole system). At each point in time t, the truth values of all ground atoms of a module are determined by a pair of models (M_t, N_t) where the partial model M_t

transformation *reflect up* : **epistemic-object**
domain *symptom interpretation*
epistemic output signature
epistemic-output-symptom-interpretation;
co-domain *hypothesis generation*
object input signature *input-hg*;
sort links (*OA*,*hypotheses*)
object links identity
atom links
(*false*(*X*:*OA*),*si-false*(*X*:*hypotheses*)):
<<**true**,**true**>, <**false**,**false**>>;
endtrans

transformation *reflect down* : **object-target**
domain *hypothesis generation*
epistemic output signature *output-hg*;
co-domain *symptom interpretation*
object input signature
target-input-symptom-interpretation;
sort links (*hypotheses*,*OA*);
object links identity;
atom links
(*possible-hypothesis*(*X*:*hypotheses*),
target(*dynamic-target,X*:*OA*,*confirm*)):
<<**true**,**true**>, <**false**,**false**>,
<**unknown**,**false**>>;
endtrans

Fig. 14 The transformations *reflect down* and *reflect up*.

represents the truth values of the object-level at point t and the complete model N_t represents the truth values of the meta-level at point t. More precisely, a *state* is described by the truth values of all object- and meta-information facts of all modules, by the truth values of all meta-information facts of all transformations, and by the truth values of all object- and meta-information facts of the *supervisor module* which interprets the global control rules.

A *state transition* from a pair of models (M_t,N_t) to (M_{t+1},N_{t+1}) can be achieved by a number of complex *transition functions* which are built into the semantics of DESIRE. Such a transition function expresses either an object-level reasoning step of a module with update of its meta-level (in the case of the supervisor module it defines the interpretation of the global control rules) or by transitions which realise object-object interaction, meta-meta interaction, upward reflection, or downward reflection. Each of these transition functions is defined (operationally) by a set of transition rules and a virtual rule interpreter which are hard-wired into the semantics of DESIRE (see Gavrila & Treur (1994) for more details).[9]

The overall semantics of the specification of a kbs is specified by a trace of a number of model pairs $(M_1,N_1),...,(M_n,N_n)$.

1.2.5 Comparing DESIRE with (ML)² and KARL

A significant difference exists at a **conceptual perspective** between DESIRE and KADS oriented specification languages like (ML)² and KARL. DESIRE tries to cover most aspects of a system specification: the reasoning behavior, the system/user-interaction, and it allows the specification of multi-agent systems. Languages like (ML)² and KARL are much more restricted as they focus on specifying the reasoning behavior of a kbs.

Apart from this general distinction, a rough analogy can be made by identifying the global control rules with the task layer and the modules with inference actions in KADS. With regard to the representation of control, (ML)² and KARL use a procedural representation whereas DESIRE uses the more conventional view in Artificial Intelligence. Control is specified by a set of rules and a rule interpreter has to evaluate preconditions of these rules and select a rule with valid preconditions. The representation of control by means of rules has the well known advantages and disadvantages of production rule systems. On the one hand, local control decisions can be represented in an elegant manner and reasoning over control decisions can be expressed. On the other hand, the global behavior of the system becomes difficult to predict.[10]

The different representation of control between DESIRE on the one hand and (ML)² and KARL on the other is a consequence of the different scopes of the approaches. The latter define the control of the reasoning process of domain and inference layers at the task layer. The former does not only aim at specifying the control of the object-level reasoning process but it also specifies reasoning processes over the control of the object-level inferences. That is, DESIRE aims to represent *strategic reasoning*. In the original KADS-I framework (Wielinga et al., 1992) this type of knowledge was located at a fourth layer in the model of

9. For example, the internal substates of the interpreter which define the semantics of a reasoning module are: *integrate new input from the input buffer, prepare task execution, work on current subtask, integrate new output into the output buffer, finish task execution, suspended, idle*. For each phase specific transformation rules are defined.

10. An intermediate position is taken by the operational knowledge specification language *MoMo* which provides Petri nets for specifying control (Voss & Voss, 1993).

expertise which was called the *strategic layer,* but most of the KADS-oriented languages do not provide modelling primitives for this layer.

The reasoning modules of DESIRE can be roughly identified with inference actions in $(ML)^2$ and KARL, but DESIRE provides a much more sophisticated means of controlling the reasoning process of an inference action. The information state is represented at a meta-level and via transformation other modules can influence the meta level facts and therefore the object-level reasoning of the module. This extended granularity of controlling the reasoning process of the elementary buildings blocks (the elementary inference actions) is not provided by $(ML)^2$ and KARL.

An explicit distinction between domain knowledge and the generic specification of the reasoning process by a problem-solving method as in KADS is not made in DESIRE. One has to admit that the module and transformation concept of DESIRE is powerful enough to cover such a distinction. Still it is not so much the question whether something could be expressed in a language but rather whether the language itself provides a strong bias on how things become expressed in it.

The descriptions of states and of state transitions is much more complex in DESIRE than in $(ML)^2$ or KARL. The powerful control of the reasoning process of a module by meta-level facts and the corresponding interaction primitives (**epistemic-assumption**, **object-target** etc.) makes understanding a DESIRE specification not an easy task. The precise role of *(possible hypothesis*(*X*:*hypotheses*), *target*(*dynamic-target,X*:*OA,confirm*)): <<**true**,**true**>, <**false**,**false**>, <**unknown**,**false**>>; in an **object-target** transformation for the reasoning process is defined by complex transition functions included into the semantics of DESIRE. There are also complex and implicit dependencies between various parts of the entire specification. For example, that the *global* control rule (3) in Figure 11 refers to the case where the module *symptom interpretation* fails can only be understood when looking at the *local* specification of the transformation *reflect down* and its target parameter *confirm* and the mapping of the truth value *unknown*.

From a **semantical point** of view a significant difference between DESIRE and $(ML)^2$ or KARL lies in the fact that the former uses temporal logics for specifying the dynamic reasoning process whereas the latter use dynamic logic. In dynamic logic, the semantics of the overall program is a binary relation between its input and output sets (M_i,M_o). In DESIRE, the entire reasoning trace T which leads to the derived output is used as semantics:

$$T = M_i, M_1, \ldots, M_n, M_o$$

Therefore, DESIRE uses a sequence of models to define the semantics of a specification. Otherwise, the truth of formulas depends on the current state only. A semantics which includes the derivation path by a sequence of models is necessary when one wants to specify *dynamic integrity constraints* on the reasoning process which do not only restrict valid relations between input and outputs but which also define restrictions for the reasoning process itself. Such constraints are often used in the specification of information systems (Jungclaus, 1993) or database updates (Bonner & Kifer, 1993). Currently, it is not clear at all whether a semantics based on model pairs as in $(ML)^2$ and KARL or a path-semantics by a sequence of models as in DESIRE is better suited for specifying kbs. An interesting feature of Transaction logic as discussed by Bonner & Kifer (1993) is that it integrates both types of semantics into one coherent framework.[11]

DESIRE provides an interpreter which can be used to **execute specifications**. Otherwise, the operationalization is naturally incomplete as first-order logic is provided as a specification language for the local specification of the modules. A further problem for the operationalization is the non-determinism provided by the **any**-parameter and the case where several control rules apply. An interpreter can realise a random choice but the outcome of testing depends on these selections and does not cover all possible cases. That is, the specified kbs can have consequences which are not given by executing one trace of its specification.

2 Specification Languages in Software Engineering

In this section we discuss specification methods used in software engineering. The main problem for such a discussion and comparison with work in knowledge engineering is caused by the overwhelming amount of work which is extant in this area. The development of formal, executable, or semiformal specification languages has almost twenty-five years of tradition in SE and has led to a multitude of languages which are impossible to keep track of.[12] A scientific evaluation of some commercial applications for formal specification techniques is provided by Craigen et al. (1993).[13]

As a consequence of the large number of approaches, we can only provide a very restricted selection of them. First, we discuss the broad and well established field of algebraic specification techniques (Bidoit et al., 1991), which provide means for a functional specification of a system. Secondly, we discuss the Vienna Development Method - Standard Language (VDM-SL) (Jones, 1990) and Z (Spivey, 1992), which describe a system in terms of states and operations working on these states. Both languages are the result of long running (and still ongoing) research projects with a multitude of commercial applications (Woodcock & Larsen, 1993). Thirdly, we discuss Evolving Algebras (Gurevich, 1993 & 1994). This more recent approach aims at closing the gap between computation methods and specification methods. The dynamic behavior of software systems is described by an algebra which becomes modified during the execution of an algorithm. We conclude the section with a comparison to semiformal specification techniques like Structured Analysis (see Yourdan, 1989).

2.1 Algebraic Specifications

Algebraic specification methods have a twenty year history and have become one of the major areas of research in theoretical computer science. An overview and bibliography of different approaches to algebraic specification of software systems can be found in Bidoit et al. (1991). Besides their use for specifying software systems they can also be used for

11. In the case of DESIRE, this type of semantics will be necessary when the language for specifying the control gets extended. At this moment, only the current termination status of a component can be asked by the control rules. Future extension of DESIRE will provide temporal operators which refer to earlier states of the reasoning process (i.e., they refer to the history of it).

12. This has also taken place in the area of information system development but to a less extent and in a shorter period of time.

13. A list of formal methods for system development is provided by Ryan & Sennett (1993), see also Gaudel (1994) and http://www.comlab.ox.ac.uk that provides a survey on formal methods available via the World Wide Web. Recent introductions to formal specification methods are provided by Turner & McCluskey (1994) and Sheppard (1995).

defining the semantics of programming languages. The large number of algebraic specification languages forces us to focus on the essence of algebraic specification techniques without discussing specific realisations in detail. First, we discuss algebraic methods for specifying the functionality of a system. Then, we discuss an example for approaches which specify the dynamic behavior of software systems. Finally, we discuss specification languages in knowledge engineering which rely on algebraic techniques.

2.1.1 Algebraic Specification Techniques for Functional Specifications: Abstract Data Types

Detailed introductions to algebraic specification methods are provided by Ehrig & Mahr (1985), Ehrig & Mahr (1990), and Wirsing (1990). The essence of algebraic approaches is the use of a *many-sorted* (or *order-sorted*) *algebra* to specify the data structure and the functionality of a program. Elementary data types are modelled by sets, each datum is modelled by an element of such a set, and functions (i.e., operators) on these sets model the functionality of the software artefact. The properties of the functions (0-ary functions model constants) are further described by a set of atoms in a logic language including equality. An *abstract data type* (ADT) is an isomorphism class of many-sorted algebras. An *algebraic data type* is the definition of an abstract data type by means of a signature (sorts and functions) and some axioms (logical formulas) that the algebraic isomorphism class must fulfil. An *algebraic specification* is a description of one or more such abstract data types by algebraic data types. Figure 15 gives the domain layer of our running example in terms of an algebraic specification.

Algebraic techniques provide several techniques to structure complex specifications like constraints, structuring operators, parameterization, and modularisation (see Ehrig & Mahr, 1990). An abstract data type could be used to define a module, which exports some of its functions and sorts and hides internal details of their definitions. It can use other abstract data types by importing them. *Parameterization* is an interesting feature in the context of kbs as it enables the generic specification of inference actions. The connection to a domain layer can be achieved by binding generic parameters to actual ones (compare section 2.1.3). Figure 16 sketches the inference action *generate*, its views and its output store. The signature and axioms of views and (if available) input stores are imported. The output store *HYPOTHESIS* imports the abstract data type *GENERATE* which defines the inference action *generate*. The connection to the domain layer can be achieved by binding the generic parameter *domain layer* to an actual domain layer (e.g., as given in Figure 15).

Roughly, three different types of semantics can be distinguished: Initial semantics, terminal semantics, and loose semantics (cf. Wirsing, 1990). In the *initial semantics*, only the terms that can be proven to be equal from the given axioms are identified. In the *terminal semantics* only those terms are different whose inequality can be proven from the axioms. The *loose semantics* regards all models of the axioms as semantics besides models which contain junk, i.e. terms which cannot be denoted by ground terms. Initial and terminal semantics are special models of the set of models under the loose semantics.

An initial semantics leads to a *unique* (up to isomorphism) algebra for an abstract data type similar to the minimal model semantics in logic programming. The so-called quotient model is analogous to the Herbrand model of a logic program. It consists of all possible closed terms of the specification, modulo provable equality. Otherwise, the initial semantics (as the closed-world assumption) has as a consequence that all non-provable equalities given a set of axioms

spec *DOMAIN*
  import *BOOLEAN*;
  sorts
    symptom, *disease*, *real, boolean*;
  functions
    no-fever, *low-fever*, *high-fever* : *symptom*;
    healthy, *influenza*, *pneumonia* : *disease*;
    0.5,0.1,0.05 : *real*;
    actual symptom : *symptom* $\rightarrow$ *boolean*;
    caused-by : *disease* $\times$ *symptom* $\rightarrow$ *boolean*;
    probability : *disease* $\rightarrow$ *real*;
    prefer : *disease* $\times$ *disease* $\rightarrow$ *boolean*;
    $>$: *real* $\times$ *real* $\rightarrow$ *boolean*;
  variables X_1, X_2 : *disease*; Y_1, Y_2 : *real*; Z : *boolean*;
  axioms
    actual-symptom(*low-fever*) = *true*;
    actual-symptom(*no-fever*) = *false*; *actual-symptom*(*high-fever*) = *false*;
    caused-by(*healthy,no-fever*) = *true*;
    caused-by(*healthy,low-fever*) = *false*; *caused-by*(*healthy,high-fever*) = *false*;
    caused-by(*influenza*,*low-fever*) = *true*;
    caused-by(*influenza*,*no-fever*) = *false*; *caused-by*(*influenza*,*high-fever*) = *false*;
    caused-by(*pneumonia,low-fever*) = *true*; *caused-by*(*pneumonia,high-fever*) = *true*;
    caused-by(*pneumonia,no-fever*) = *false*;
    probability(*healthy*) = 0.5;
    probability(*influenza*) = 0.1;
    probability(*pneumonia*) = 0.05;
    $prefer(X_1,X_2) = Z \leftrightarrow probability(X_1) = Y_1 \wedge probability(X_2) = Y_2 \wedge (Y_1 > Y_2) = Z$;
endspec

Fig. 15 An algebraic specification of the domain layer.

are assumed to be non-equal. Loose semantics implies that a specification and its axiom set describes a set of possible implementations (each valid model corresponds to an implementation). That is, a specification *abstracts* from details which are fixed later on during the implementation. An implementation must only fulfil the equalities which are explicitly specified by the axioms. This freedom is lost when applying initial or terminal semantics. For example, when a specification defines a set, an implementation can realize this set by a list with according list operators. Neither how this list is organized nor the fact that a list is used to implement the set are of any interest when specifying the set. The only requirement is that the implementation provides all the necessary functionality which is specified. In this way a specification thus defines a whole class of implementations which behave identically for all specified aspects and which may behave differently for aspects which are not determined by the specification.

Based on the initial semantics, term rewriting techniques can be used to *execute* algebraic specifications. That is, formal specification can be evaluated by testing. Execution is simulated by deriving equalities of terms. More complex proofs require theorem proving techniques for logics with equality. An example of an operational algebraic specification language based on term rewriting is OBJ (Futatsugi et al., 1985; Gerrard et al., 1990).[14]

14. See Bidoit et al. (eds.) (1991) for a survey of further approaches.

IA-spec *GENERATE*
  import *FINDINGS, CAUSALITY;*
  functions
    generate : *hypotheses* → *boolean*;
  variables
    X : *hypotheses*; *Y* : *findings*; *Z* : *boolean*;
  axioms
    generate(X) = Z
      ↔ *causality*(*X*,*Y*) = *Z* ;
end-IA-spec

VIEW-spec *FINDINGS*(*domain layer*)
  parameter *domain layer;*
    sorts *findings*;
    functions *finding* : *findings*;
end-VIEW-spec

VIEW-spec *CAUSALITY* (*domain layer*)
  parameter *domain layer;*
    sorts *hypotheses*, *findings*, *boolean*;
    functions
      causality : *hypotheses* × *findings*
        → *boolean*;
end-VIEW-spec

STORE-spec *HYPOTHESIS*
  import *GENERATE;*
  functions
    current-hypotheses : *hypotheses* → *boolean*;
  variables *X* : *hypotheses*;
  axioms
    current-hypotheses(*X*) = *generate*(*X*);
end-STORE-spec

Fig. 16 An algebraic specification for the inference action *generate diagnosses*.

2.1.2 Specifying the Dynamics of a Software Artefact with Algebraic Techniques

In the following, we discuss *dynamic equational logic* (Wieringa, 1991a) as an example of approaches to representing states and state changes in an algebraic framework. Dynamic equational logic is used to define a formal semantics for the Conceptual Model Specification Language (CMSL) which is an object-oriented specification language for information systems (see Wieringa & van de Riet, 1990).

Wieringa (1991a) examines the usefulness of algebraic specification techniques (i.e., order-sorted equational abstract data types) for the specification of object-oriented databases. An abstract data type can be used to specify an object. Methods can be specified by operations (i.e., functions). A significant problem in this context occurs from the fact that an object has a state which can change over time. ADTs have neither a notion of state nor an explicit representation of state transitions. Wieringa (1991a) introduces the following interpretation and extension of ADT to capture the notion of states and state transitions. Figure 17 gives an example of a simple abstract data type. There are many possible functions that can be used as

spec *PERSON-OBJECT*
  sorts *persons, natural numbers, events;*
  constants *p* : *persons;*
  attribute
    age : *persons* → *natural numbers*;
  events
    increment : *persons* → *events*;
endspec

Fig. 17 A simple ADT.

interpretations of the attribute *age* declared in the object specification, for instance, *age*(*p*) = 20 and *age*(*p*) = 30. Each such interpretation is a *possible world* and represents one possible state of the object.

State transitions are modelled by a specific sort *events* and operations which have this sort as co-domain. These operations are interpreted as programs in dynamic logic (Harel, 1984;

Kozen, 1990). Dynamic logic defines Kripke structures providing a set of possible worlds and interprets programs as a binary relation between possible worlds. Given the signature of our example in Figure 17, *increment*(p) is the only available program. The modal operators [.] and <.> can be used to define constraints which characterize the behavior of this program. The following constraint enforces that the *age* of the person p in the possible world after the execution of the program must be incremented by one compared to the value in the possible world before the program execution.

$$age(p) = n \rightarrow [increment(p)]\ age(p) = n + 1$$

States are introduced in ADT by regarding each possible interpretation of an ADT as a possible world (i.e., a state). Notice, that *one* state corresponds to *one* model of the algebra. Transitions between states are binary relationships over these interpretations and dynamic logic is used to characterize these transitions further. Control is defined in a non-constructive manner by constraints used to restrict all possible state transitions.

2.1.3 Algebraic Approaches in Knowledge Engineering

Algebraic specification methods are also used for specifying kbs. Algebraic specification methods were mainly designed for functional specification and not for the specification of the dynamic (reasoning) behavior of a system. Therefore, we will focus our attention on this problematic aspect.

Nakagawa et al. (1993) presents the algebraic specification of a kbs for simple scheduling tasks with OBJ (Futatsugi et al., 1985; Gerrard et al., 1990). The main problem with this formalization is the representation of the dynamic reasoning behavior of the system which is implicitly encoded in the axioms and their term structure and realized by the term rewriting technique of OBJ.

The knowledge specification language K_{BS}SF (in`t Veld,1993; Spee & in`t Veld,1994), which was developed as part of the ESPRIT project VITAL, uses algebraic specification techniques for specifying the domain and inference layers of a model of expertise. The domain layer and every inference action are specified by modules defined by ADT's. The logical language for formulating axioms is order-sorted first-order logic. The connection between modules specifying domain knowledge and modules specifying generic inference knowledge is achieved by *parameterization*. A procedural language on top of the algebraic specification is used to express control over the dynamic reasoning process. Sequence, alternative, and iteration of module executions can be expressed. Spee & in`t Veld (1994) define a formal semantics for the entire language for the case where the logical language used in the algebraic specifications is restricted to Horn logic. Atomic programs are the execution of a module specified by an ADT. The usual minimal Herbrand model semantics is applied as semantics for the modules. The execution of a module delivers its minimal Herbrand model as output. The entire semantics is defined in terms of transitions between configuration in a Plotkin style (Plotkin, 1981). A *configuration* is characterized by the current values of the program variables (i.e, stores) which are changed by the execution of a module and a stack of further control statements which describe the part of the program which has not been executed. A *transition* executes the next statement in the stack of further control statements. As a result the current values of program variables are changed and the stack is updated. Besides some minor details, this semantics of K_{BS}SF is identical to a combination of the declarative semantics of domain and inference layers in KARL (see Fensel (1995), who

$$\begin{array}{l}
\cup : process \times process \rightarrow process \\
\delta : process \\
; : process \times process \rightarrow process \\
* : process \rightarrow process \\
\hline
(p \cup q) \cup r = p \cup (q \cup r) \\
p \cup q = q \cup p \\
p \cup p = p \\
p \cup \delta = p \\
(p \cup q); r = (p;r) \cup (q;r) \\
r; (p \cup q) = (r;p) \cup (r;q) \\
p^*; q = q \cup p; p^*; q
\end{array}$$

Fig. 18 Algebraic specification of the choice operator in TFL.

defines a declarative semantics for KARL) and the operational semantics of the task layer in KARL (see Angele (1993), who defines an operational semantics for KARL).

A uniform approach for the algebraic specification of static and dynamic aspects of a kbs is provided by the knowledge specification language TFL (Pierret-Golbreich & Talon, 1995) which adapts the algebraic specification language PLUSS (cf. Gaudel, 1984; Bidoit, 1989) for this purpose. ADT's are applied to specify domain and inference knowledge and loose semantics is applied. As in $K_{BS}SF$ parameterization is applied to connect the generic modules expressing inference knowledge with modules specifying the domain knowledge. Procedural control is specified by so-called process modules which incorporate the control expressions as operations into the framework of ADT's. The test operator for formulas (*?*), sequence (;), nondeterministic choice ($\cup$), and iteration (*) are specified as functions and axioms are used to specify these operators further. In addition to this procedural definition of control, primitives for strategic knowledge are provided. A *focus* can be used to privilege a subset of all processes and to choose between data- and goal-driven control. A *strategy* can be used to activate or to combine foci or substrategies. The recursive definition of strategies enables multiple meta-levels as in DESIRE for defining strategic reasoning. Figure 18 provides an example for the algebraic definition of control by picturing the axioms used in TFL to specify the choice operator.[15] TFL does not provide the notion of a *state*. Compared to $K_{BS}SF$ it has the advantage of integrating the control aspect into the algebraic setting which prevents it from the paradigm shift between the static and dynamic part of a specification in $K_{BS}SF$.

2.2 The Vienna Development Method and Z

The Vienna Development Method VDM (cf. Jones, 1990; Andrews & Ince, 1991) is a *method* for the formal specification and development of software systems. It consists of a specification language called VDM-SL, rules for data and operation refinement, and a proof theory for conducting properties of the specifications and the correctness of refinement steps. Z (cf. Spivey, 1988; Spivey, 1992; Wordsworth, 1992) is a formal specification *language* for specifying software systems. Spivey (1992) defines a standard for Z. A definition of its mathematical semantics is provided by Spivey (1988). Both approaches have been used for a large variety of different problems over more than a decade. Meanwhile a series of joint workshops has been organized (e.g. Bjorner et al., 1990). A common introduction to VDM and Z is given by Sheppard (1995), a comparison of both languages is provided by Hayes

15. Actually, this is the way in which (concurrent) processes and their dynamic interaction are specified in *process algebra* (Baeten & Weijland, 1990; Baeten, 1990).

(1992) and Hayes et al. (1994). As both specification languages fall in the same general class (called *model*-oriented as distinct to algebraic approaches which are called *property*-oriented), we will discuss one of them in more detail (we have chosen Z) and point out only some differences for the other. The application of Z for specifying kbs are described by Milnes (1992) and Krause et al. (1993).[16]

2.2.1 Z[17]

Z (Spivey, 1988 & 1992) describes the data, their interrelationship, and the functional behavior of a system using sets, relations, and functions. Z is based on typed set theory. Static and dynamic aspects of a system are uniformly described by so-called *schemas*. Complex specifications can be built up by combining several of these schemas.

A Z specification consists of a sequence of definitions and theorems. *Definitions* can be categorized as: *types*, *axiomatics*, or *schemas*. Types can be elementary sets, functions, relations, sets of sets etc. The mathematical toolkit of Z provides a library of predefined types, functions, and relations. Further types can be introduced by defining new basic types or by combining already given types. Axiomatics can be used to define global variables together with their type and invariants. Significant for Z are *schemas* which can either be used to characterize the state space and invariant relationships on states of a system or operations which change these states. The former are called *state schemas* and the latter are called *operation schemas*. In the following, we discuss some aspects of the schema calculus of Z which seem to be relevant in our context. We will use the specification of a stack as illustration for the schema calculus of Z. Later on we will show how it can be used to model a kbs for our simple diagnostic task.

Schemas

The general form of a schema is given in Figure 19. S is the name of the schema, D defines

S
D
P

Linear notation: $S = [D \mid P]$

Fig. 19 The general form of a schema in Z

the signature, and the predicate P defines invariant relations. The signature D consists of a list of typed variables $X_1 : T_1,..., X_n : T_n$ and P is a formula which defines relationships which must hold between the values of the variables. An example of a state schema is given in Figure 20. It defines the state space of a stack of natural numbers. The global variable *stack-max* is defined by an axiomatic which defines an upper-limit of twenty for the population of the stack. Sequence is a built-in type constructor of Z.

Stack-state
stack : **sequence N**
$0 \le \|stack\| \le stack\text{-}max$

stack-max : **N**
stack-max = 20

Fig. 20 The state space of a simple stack in Z.

16. See van Harmelen & Fensel (1995) for further examples.
17. See also http://www.comlab.ox.ac.uk/archive/z.html.

Generic Schemas

It is possible to have generic schemas where formal parameters $X_1,...,X_n$ can be used in type definitions. When the generic schema is used later on, the formal parameters become instantiated by types $T_1,...,T_n$:

$$S_I == S[T_1,...,T_n]$$

The definition of the state space of a generic stack is therefore

$$\mathit{Stack\text{-}state}[X] = [\mathit{stack} : \mathbf{sequence}\ X \mid 0 \leq |\mathit{stack}| \leq \mathit{stack\text{-}max}]$$

Schema Inclusion

The *schema calculus* provides operators for combing schemas. The above instantiation of a generic schema already provides an example. Further operators are discussed during the following. A schema $S_1 = [D_1|P_1]$ can use another schema $S_2 = [D_2|P_2]$ as part of its definition by inclusion:

$$S == [D_1, S_2 \mid P_1] = [D_1 \cup D_2 \mid P_1 \wedge P_2]$$

Schema decorations provide means to specify operation schemas.

- $S' == [D|\ P]'$ with $D = X_1 : T_1,..., X_n : T_n$ is defined by replacing each variable X_i in D and P by X_i'. S denotes the state *before* executing an operation and S' denotes the state *after* executing an operation. The operation schema which describes the operation has to include both schemas.
- $S? == [D|\ P]?$ with $D = X_1 : T_1,..., X_n : T_n$ is defined by replacing each variable X_i in D and P by $X_i?$. $S?$ denotes an input signature of an operation schema which includes $S?$.
- $S! == [D|\ P]!$ with $D = X_1 : T_1,..., X_n : T_n$ is defined by replacing each variable X_i in D and P by $X_i!$. $S!$ denotes an output signature of an operation schema which includes $S!$.

The *schema notation* Δ and Ξ are short-cuts for the inclusion of state schemas by an operation schema.

- $\Delta S = [S,S' \mid\ ]$. An operation schema which changes the state described by the schema S has to include the schema ΔS (instead of S,S').
- $\Xi S = [S,S' \mid X_1 = X_1',..., X_n = X_n']$ with $S = [X_1 : T_1,..., X_n : T_n \mid P]$. An operation schema which does *not* change the state described by the schema S has to include the schema ΞS (instead of $[S,S' \mid X_1 = X_1',..., X_n = X_n']$).

Figure 21 defines four operations *Init, Push*, *Pop*, and *Top* for our stack example (see Sheppard, 1995). *Init* initializes the stack with the empty sequence, *Push* gets the input i from type $\mathbf{N}$, forms a sequence from it and concatenates it with the stack. *Pop* removes the head item from the stack. *Push* and *Pop* cause a state change. *Top* returns a copy of the head of the stack as output but does not change the state of the stack.

Schema Composition

Finally, *schema composition* can be used to express the *sequence* of two operation schemas (which should not be mixed with the built in data type sequence). The basic idea behind schema composition is that the schema expression:

$$P == Q\ ;\ R$$

defines a new schema P such that if Q can bring about a state change from S_1 to S_2 and R can bring about a state change from S_2 to S_3 then the schema P can bring about a state change

Push

$\Delta Stack\text{-}state$
$i? : \mathbf{N}$

$stack' = \langle\langle i?\rangle, stack\rangle$

Pop

$\Delta Stack\text{-}state$

$stack' = \mathbf{tail}\ stack$

Top

$\Xi Stack\text{-}state$
$o! : \mathbf{N}$

$o' = \mathbf{head}\ stack$

Init

$Stack\text{-}state'$

$stack' = \langle\rangle$

PrePush

$Stack\text{-}state$
$i? : \mathbf{N}$

$|stack| < stack\text{-}max$

Fig. 21 Operation schemas of a simple stack in Z.

from S_1 to the state S_3.

Proof Obligations and Theorems

Schema preconditions together with the precondition calculus can be used to derive and to express the preconditions of an operation schema. That is, it describes all states upon which the operation can be successfully carried out. An example is given in Figure 21 for the operation schema *Push* by the schema *PrePush* which expresses the preconditions of *Push*. A Z specification consists not only of definitions of types, states, and operations. Theorems are introduced into Z specifications to show that initial states are legal, to derive preconditions for operation schemas or to show that the specification enjoys certain properties. For example, a proof obligation of the stack example given in Figure 21 is to show that the initial state of the stack resulting from the operation *Init* satisfies the state invariant (defined in Figure 20). A specification of states and operators leads to a number of proof obligations which must be fulfilled in order to complete a specification in Z.

Specifying A Diagnostic KBS With Z

In Figure 22, we give a specification of our running example in Z. We use axiomatics to define the domain layer as it does not change during the reasoning process. Generic state schemas are used to model knowledge roles. The use of state schemas to model knowledge roles is quite natural for dynamic knowledge roles (i.e., stores in KARL) which represent the state of the reasoning process. Static knowledge roles (i.e., views in KARL) which provide access to domain knowledge could also be modelled by axiomatics as they model static relationships. The reason for modelling them by states lies in the fact that Z provides generic schemas but no generic axiomatics. Operation schemas are used to model inference actions. Their schemas have to include the schemas which represents their knowledge roles. Only the internal knowledge roles *Causality*, *Hypothesis*, and *Preference* have to be modelled by schemas. Findings are modelled by the input variable *finding* of *Generate* and diagnoses are modelled by the output variable *diagnosis* of *Select*. The mapping from the generic terminology of the inference layer onto the domain layer is achieved by instantiating the formal generic parameters of the schemas, that model knowledge roles, with actual ones.

We could also define the control at the task layer by a simple sequence of the two operation schemas *Generate* and *Select*. In general, Z does not provide appropriate means for expressing procedural control over the execution of operation schemas. It is possible to model sequence and choice (by means of if-then-else) but no loops (or recursion) constructs are provided.

Domain Layer

disease ::= healthy | influenza | pneumonia
symptom ::= no-fever | low-fever | high-fever

probability : disease $\rightarrow$ $\mathbf{R}$

probability(healthy) = 0.5 $\wedge$ probability(influenza) = 0.1 $\wedge$ probability(pneumonia) = 0.05

caused-by : disease $\times$ symptom

caused-by(healthy,no-fever) $\wedge$ caused-by(influenza, low-fever) $\wedge$ caused-by(pneumonia, low-fever) $\wedge$ caused-by(pneumonia, high-fever)

Inference Layer

Generate

Δ*Mapped-Hypotheses*
Ξ*Mapped-Causality*
finding? : X_2
$x : X_1$

$x \in \{hypotheses'\} \Leftrightarrow (x, finding?) \in causality$

Causality$[X_1,X_2]$

causality : $X_1 \times X_2$

Hypotheses$[X]$

hypotheses : $\mathbf{P}(X)$

Select

Ξ*Mapped-Hypotheses*
Ξ*Mapped-Preference*
diagnosis! : X
$x : X$

$diagnosis! \in hypotheses \wedge$
$\neg x\ (x \in hypotheses \wedge (diagnosis!, x) \in preference)$

Preference$[X]$

preference : $X \times X$

Mapping Domain Layer - Inference Layer

Mapped-Preference

preference : disease $\times$ disease
D_1, D_2 : disease

$(D_1, D_2) \in preference$
$\Leftrightarrow$ probability(D_1) < probability(D_2)

Mapped-Causality == *Causality*[disease, symptom]
Mapped-Hypotheses == *Hypotheses*[disease]

Task Layer: *task* == *Generate*; *Select*

Fig. 22 A simple diagnostic example in Z.

Executability

Z is a formalization language which does not aim at operationalization. However, the languages SETL (cf. Schwarz et al., 1986; Doberkat & Fox, 1989) and PROSET (Hasselbring, 1994), which are executable specification languages based on set theory, are close to Z. They provide additional control constructs and are restricted to *finite* set theory. Actually, Z has been used to define the formal semantics of SETL and a subset of PROSET.[18]

---

18. The features of PROSET which caused difficulties for the specification in Z result from the combination of data persistency with parallelity and exception handling (personal communication with E.-E. Doberkat).

2.2.2 VDM-SL

VDM-SL[19] (Jones, 1990; Andrews & Ince, 1991) describes a system by means of a data model, which defines its possible states, together with a set of operations, which express the required behavior of a system. Each operation is defined as a relation between input and output values of various defined types. Preconditions, post-conditions, and invariants are means for specifying these data models and operations. VDM provides a number of proof obligations which can be used to show the mathematical self-consistency of a specification. Subsequently, we sketch only some significant points of VDM.

The VDM Modules

The general schema of the specification of a module is given in Figure 23. As the module concept of VDM is rather straightforward, we will not discuss the parts which are concerned with the relationships of the module with its environment (**parameters, import from, instantiation, exports**), but focus on the internal part introduced by **definitions**. Here, the most interesting parts in our context are **state** and **operations**. It should be mentioned that functions can also be defined in an implicit manner as we will introduce for operations in the following. In both cases, proof obligations arise which must ensure that an implicitly specified function is satisfiable.

module VDM-module
parameters
import from
instantiation
exports
definitions
    types
    values
    state
    functions
    operations
end VDM-module

Fig. 23 The module in VDM-SL

The module concept allows specifications to be structured as with the use of schemas in Z. However, VDM does not provide an analogue of Z's schema calculus.

States, Dynamics, and Proof Obligations in VDM

A *state* is described by an identifier, typed state variables which describe parts of the state, invariants, and initial values for the state variables. A first proof obligation is to prove that the initial values satisfy the state invariants and the types of the state variables.

The definition of an *operation* includes the state variables which are accessible by it. One can distinguish between read-only and write/read access. In addition, a *definition* of the operation must be provided which describes the functionality of an operation by a pair of *pre-* and *postconditions*. These implicit definitions of operations provide an abstraction mechanism as they describe a whole class of possible implementations. Pre-conditions represent *assumptions* which can be made by an implementation and post-conditions define *obligations* which must be met by it.

The proof obligations of VDM requires the proof that the post-condition can be fulfilled if the precondition is satisfied. From a logical point of view this proof corresponds to the derivation

19. See also *The VDM Bibliography*, http://www.ifad.dk/pub/docs/vdm.html.

of the precondition schema *preS* for a schema *S* in Z. From a methodological point of view two arguments can be made. First, the explicit distinction of pre- and post-conditions provides a better formulation of the proof obligation at a conceptual level. Second, this distinction can be used as a guideline during the process of writing the specification: “there is also a pragmatic point: in reading many industrial (informal) specifications, I have observed that people are actually not so bad at describing what function is to be performed; what they so often forget is to record the assumptions” (Hayes et al., 1994). An introduction to proofs in VDM is provided by Bicarregui et al. (1994) who define an axiomatization, inference rules, and informal proof guidelines for VDM-SL.

Language constructs to express control over the execution of substeps are introduced in VDM-SL to support stepwise development of implementations from formal specifications. That is, they are provided for the design activity: Data reification supports the transition from abstract to concrete data types and *procedural control constructs as sequence, alternative,* and *loops* are provided to decompose the specifications of operations. Such an explicit definition defines an operation in an algorithmic style whereas an implicit definition of operations by a pair of pre- and postconditions does not refer to how the functionality of such an operation is achieved.

Executability

VDM-SL and its predecessor languages have been designed as formal specification languages. More recent approaches to VDM developed interpreters for subsets of the language (cf. Lassen & Larsen, 1991; Andersen, 1992; Elmstrøm et al., 1994). Since the full VDM-SL language is not executable in general, these interpreters have to exclude language features like infinite sets, execution of type binding, purely implicitly defined functions and operations. A further problem for the operationalization effort is caused by loose specification. In the non-deterministic (or underdeterminedness) case, all states must be regarded simultaneously; or the interpreter must choose a state and the operationalization is thus incomplete. In both cases, prototyping (i.e., evaluation by testing) becomes a very complicated matter. The outcomes either depend on arbitrary choices of the interpreter or all possible outcomes have to be evaluated by the software engineer. Actually, Hayes & Jones (1989) use these problems with the execution of non-deterministic specifications as the main argument against executable specifications.

2.3 Representing a State by an Algebra: Evolving Algebras

In the following, we discuss approaches which use an algebra to characterize the state of a program and the execution of a program by changes of this algebra.

Evolving Algebras (*ealgebras*) (Gurevich, 1993 & 1994) provide a means for defining an operational semantics for algorithms.[20] As with Turing machines, ealgebras aim at defining a mathematical framework for describing and simulating arbitrary algorithms. However, this should be achieved without encoding such an algorithm at the very low level of abstraction which Turing machines require.

In the following we briefly introduce the definitions of sequential (deterministic) ealgebras. In the example, we have to use parallelism which will be discussed then. We refer the reader

20. See E. Börger: *Annotated Bibliography on Evolving Algebras*, available via ftp apollo.di.unipi.it, pub/Papers/borger and http://www.eecs.umich.edu/ealgebras.

to Gurevich (1994) for the precise definition of sequential ealgebras and their extension to nondeterminism, parallelism, distributed algorithms (i.e., multi-agent systems), and real-time algorithms. The two basic description mechanisms of ealgebras are *states* and *state transitions*.

Algebras are used to model states. One *state* is modelled by one static algebra. A set of states (i.e., possible worlds) is provided by a set of similar algebras (i.e., with the same signature). In the simplest case, a signature Υ is a finite collection of function names with given arity (0-ary function names model constants). This common signature of these algebras defines the invariant of all regarded states. A static algebra of a signature Υ is a nonempty set S together with interpretations on S of function names in Υ. Such a static algebra defines one possible world (i.e., possible state). Possible worlds differ in the interpretation of the function names ofthe signature.

Transitions between states can be expressed by *function updates* of the form

$$f(t_1,...,t_r) := t$$

The two static algebras which model the states before and after the function update are identical apart from the interpretation of the function f for the argument $(t_1,...,t_r)$ which is interpreted by t in the successor state.[21] In the sequential case all terms $t_1,...,t_r,t$ have to be ground. These updates can be enriched by guards which express preconditions for their application. A *guarded update* is a rule

if b_0 **then** u_0
elseif b_1 **then** u_1
...
endif

where the b_i are the preconditions (the guards) and u_i are the updates.

A program in evolving algebra is a set of such guarded updates. A state transition is achieved by executing all guarded updates in parallel. If two updates contradict, no update is effected.

Evolving algebras should be useable for specifying arbitrary algorithms. Therefore they do not incorporate any conceptual modelling bias as languages do which restrict themselves to a specific type of software. However specifications in ealgebras can be structured and hierarchically levelled specifications can be written. External functions can be used to hide details (see Börger et al. (1994) for an example) or to represent the interaction with the environment of the system. The generality of ealgebras shows up by their use to specify programming languages like C, PROLOG, and VHDL, to validate implementations of programming languages such as PROLOG and Occam, and to validate distributed communication protocols, etc.

Figure 24 provides the specification of our running diagnostic example as an ealgebra. The domain knowledge is expressed by the initial state. The mappings from the domain to the inference layers as well as each inference action are represented by guarded updates. The sequence of the two inference actions which is defined at the task layer is encoded into the guards of these updates. The parallel version of ealgebras is required because an inference

21. Function updates can also be used to express the creation of new objects by introducing a function U with values $\{true, false\}$ and interpreting $U(a) = true$ as existence of the object a (see Gurevich, 1993).

Domain layer

diseases= {*healthy*, *influenza*, *pneumonia*}
symptoms = {*no-fever*, *low-fever*, *high-fever*}
caused-by(*healthy,no-fever*) = *true*;
caused-by(*healthy,low-fever*) = *false*; *caused-by*(*healthy,high-fever*) = *false*;
caused-by(*influenza,low-fever*) = *true*;
caused-by(*influenza,no-fever*) = *false*; *caused-by*(*influenza,high-fever*) = *false*;
caused-by(*pneumonia,low-fever*) = *true*; *caused-by*(*pneumonia,high-fever*) = *true*;
caused-by(*pneumonia,no-fever*) = *false*;
probability(*healthy*) = 0.5; *probability*(*influenza*) = 0.1; *probability*(*pneumonia*) = 0.05;

Mapping domain and inference layers

(1) **Var** X_1, X_2 **range over** *diseases, symptoms*
if $mode = init_1 \wedge caused\text{-}by(X_1, X_2)$
then $causality(X_1, X_2)$:= true, $mode := init_2$

(2) **if** $mode = init_2$ **then** *hypotheses* := *diseases*, $mode := init_3$

(3) **Var** X_1, X_2 **range over** *diseases*
if $mode = init_3 \wedge probability(X_1) > probability(X_2)$
then $preference(X_1,X_2)$ = true, *mode* := *hypothesis-generation*

Inference and task layer

(4) **Var** X **ranges over** *hypotheses*
if $mode = hypothesis\text{-}generation \wedge causality(finding,X)$
then $hypothesis(X) := true \wedge mode := hypothesis\text{-}selection$

(5) **choose** X_1 **in** *hypotheses* **satisfying** $hypothesis(X_1) \wedge \neg\exists X_2\ (hypothesis(X_2) \wedge preference(X_1,X_2))$
if *mode* = *hypothesis-selection* **then** $diagnosis(X_1) := true \wedge mode := stop$
endchoose

Figure 24 Specifying the diagnostic example with evolving algebras

action can define the modification of a function at several locations in parallel. The inference action generate which is represented by transition rule (4) derives all hypotheses in parallel which can explain the given finding. finding is an external function providing input from an user.

2.3.1 Operationalization and Formalization of Evolving Algebras

Ealgebras describe algorithms in an operational manner which easily provides prototyping as a means for evaluating such specifications. Interpreters for ealgebras enabling executable specifications are described by Huggins (1993), Kappel (1993), and Beckert & Posegga (1995). Again, the usual problems of executing nondeterministic specifications appears and as an ealgebra could include arbitrary (non-computable) external functions executability cannot always guaranteed.

Ealgebras provide an "informal" specification style as in the usual style of writing and proving theorems in mathematics. No typing with attached type checking or axiomatic semantics with attached proof calculus are defined. A formal definition of syntax and semantics and a partial axiomatization of ealgebras based on the *Modal Logic of Creation and*

Modification (MLCM) are described by Groenboom & Renardel de Lavalette (1995). MLCM (Groenboom & Renardel de Lavalette, 1994) has been developed to provide a partial formalisation and axiomatisation of the dynamic part of the *Common Object-oriented Language for Design, COLD* (Feijs et al., 1989; Feijs & Jonkers, 1992).[22] States are described by an algebra containing sorts, predicates, and functions. MLCM provides three atomic program statements to express state changes:

- NEW *c* (creating a new object and letting the constant *c* refer to it);
- $f(t_1,...,t_r) := t$ (changing the value of the function *f* on the arguments $t_1,...,t_r$ to the value of *t*); and
- $p(t_1,...,t_r) :\leftrightarrow A$ (changing the value of predicate *p* on the arguments $t_1,...,t_r$ to the truth value of *A*).

These atomic programs can be used to build up complex programs by the usual operators of dynamic logic: test operator for formulas (*?*), sequence (;), nondeterministic choice ($\cup$), iteration (*), and the modal operators [.] and <.>. Therefore, state transitions are described in a procedural manner as in dynamic logic. The difference with dynamic logic lies in the fact that a state is not represented by value assignments of program variables but by an algebra. A subset of these primitives can express control where required for the formalization of ealgebras.

2.4 Semiformal Approaches

> "It seems sound to conjecture that this kind of specifications (i.e., formal specifications) will not be put to practical use as they are: the mathematical background is still too visible." (Gaudel, 1990)

A difference when comparing formal specification languages originating from software engineering with knowledge specification languages is that the latter are generally subject to a stronger conceptual model of the system to be described. Each of them distinguishes different types of knowledge and captures the object-meta relationships between them (i.e., one type of knowledge controls the use of another type of knowledge). Thus, knowledge specification languages are much closer to the conceptual and informal or semiformal descriptions of expertise than general purpose languages from software engineering. ADT, VDM and Z define a rigorous syntactic and (several years later) semantical framework to describe systems mathematically but have been developed without an eye on semiformal specification techniques. Therefore, there is neither a smooth transition from semiformal to formal specifications nor is there any conceptual model like the model of expertise which underlies a specification. In fact, a system is specified as a structured description of a partial function, which is a very low (i.e., abstract) description.

On the other hand, as a result of the effort required to put formal techniques into practice, several authors have developed combinations with semiformal specification techniques like structured analysis (see Yourdan, 1989) or object-oriented analysis (see Coad & Yourdan, 1991). Approaches for VDM are provided by Elmstr∅m et al. (1993), Larsen et al. (1991) and Larsen et al. (1993); Randell (1990), Stepney et al. (1992) and He (1995) provide this for Z; and France & Docker (1989), Doberkat (1994) and Bourdeau & Cheng (1995) for algebraic

22. COLD is a wide-spectrum specification language which enables the specification of static and dynamic aspects, and was developed at PHILIPS Research, The Netherlands. See also http://www.cs.rug.nl/~rix/cold.html.

techniques.[23] Therefore, we will compare the conceptual model of a standard technique of software engineering based on semiformal specification techniques with the KADS model of expertise, which is used as a conceptual model by most knowledge specification languages. We have chosen to discuss approaches based on Structured Analysis (Ross, 1977; Yourdan, 1989) as it is widely used in software engineering and as it is also the point of reference for most of the above mentioned approaches, which combine semiformal and formal techniques.[24] Structured Analysis is a method which contains several informal instruments for software specifications: a *data dictionary* and *entity-relationship diagrams* are used to describe the static system aspects, i.e., the structure and domains of the data used. *Dataflow diagrams* describe the functional behavior of a system. The whole system is decomposed into individual processes, and their interactions are represented by sharing data. Dataflow diagrams consist primarily of processes, stores, and the dataflows between them. They can be hierarchically refined to allow a representation with different grain sizes. *Process-specification* techniques allow the description of the behavior of elementary processes which are not refined further by other dataflow diagrams. As dataflow diagrams should not be used to define the control flow between processes, a separate instrument for representing the *control flow* between the processes is added. Fensel et al. (1993) define a formal and operational semantics for the modelling primitives of Structured Analysis by means of KARL to get a better insight into the different conceptual models. In the following, we will present the main results of this case study (compare also Schreiber et al., 1993).

Task Layer. The task layer of KARL and the state-transition diagrams of structured analysis correspond roughly because both are means of specifying the control flow of a system.

Inference Layer. Schreiber et al. (1993) compares structured analysis and KADS and points out a significant distinction between dataflow diagrams with process specifications on the one hand and an *inference layer* on the other hand:

> "A first difference between the functional description in KADS and in conventional software engineering is that the data elements in inference structures do not refer directly to elements of the data model ..." (Schreiber et al., 1993)

The input data, intermediate data, and output data of a problem-solving process in KADS are presented by roles. Roles connect an inference layer with a domain layer. As already discussed, KARL contains three different types of roles.

- *Views*: A view reads knowledge and data from the domain layer which serve as input for an inference action.
- *Terminators*: A terminator writes results of the problem-solving process back to the output-data part of the domain layer.
- *Stores:* An intermediate role of an inference layer collects output from inference actions and provides input for other inference actions. Such a store has no connection with the domain layer. It is an instrument for modelling the dataflow dependencies between inference actions.

When comparing these three types of roles with dataflow diagrams, the difference becomes obvious. Stores in KARL correspond to stores in Structured Analysis. Terminators roughly

23. See Semmens et al. (1992) for a comparison of some approaches on combing semiformal and formal methods in SE.
24. A comparison of the KADS model of expertise and the object-oriented modelling technique OMT (Rumbaugh et al., 1991) from SE is provided by Schreiber et al. (1993).

correspond to so-called terminators in Structured Analysis.[25] But in Structured Analysis there is *no* instrument which corresponds to *views* which read *knowledge* from a domain layer. Thus, KARL inference structures correspond to dataflow diagrams, but extend them by the mapping rules in view bodies. These views deliver *knowledge* for the inference process and enable the generic, i.e., *domain-independent, specification* of the inference process and therefore its reuse in different domains.

Domain Layer. This distinction between dataflow diagrams and inference structures necessarily leads to the third issue. Is the domain layer identical to the data model in Structured Analysis? The KARL formulation of a specification of a small lending-library system which used Structured Analysis ended up with an empty domain layer (see Fensel et al., 1993). The terminology, i.e. the syntactical structure of the dataflows, is defined by the class and predicate definitions of the roles of the inference layer. That is, the data dictionary in Structured Analysis corresponds to the terminological definitions of the domain layer. Taking a closer look at the purpose of a domain layer in KADS, it becomes obvious that *the domain layer does not have a direct counterpart in Structured Analysis*. The domain layer of a knowledge-based system is used to model the domain-specific *knowledge* which is necessary to solve the task:

> "Domain knowledge can be viewed as a declarative theory of the domain. In fact, adding a simple deductive capability would enable a system in theory (but, given the limitations of theorem-proving techniques, not in practice) to solve all problems solvable by the theory." (Wielinga et al., 1992)[26]

Summary. The comparison of KARL with Structured Analysis shows significant distinctions between the approaches: first, the inference and task layers specify the problem-solving process in a domain-independent manner. Second, the domain layer contains a large amount of domain-specific knowledge required by the knowledge-based system to solve the given tasks and has no direct counterpart in Structured Analysis. Third, the inference layer and task layer add knowledge to achieve efficiency of the problem-solving process. Efficiency of the problem-solving process (i.e, the behavior of the system) is not regarded as a main issue in Structured Analysis as it should provide only a functional specification of the system. The specification of algorithms and data structures for its efficient computation is regarded as an issue during design and implementation of the system.

3 Mutualities and Distinctions

In this paper we discussed and compared specification languages from knowledge and software engineering. This section summarizes the comparison of the different languages by introducing criteria which concern two different aspects of these languages: their purpose and the possibility of expressing a conceptual model like the KADS model of expertise.[27] We do not aim at a neutral comparison, instead we take the needs which arise when specifying kbs as a gold standard.

25. That is why these names were chosen in KARL.
26. I do not completely agree with this point of view as the inference and task layers usually introduce nonmonotonic effects which extend the scope of the declaratively specified domain theory.

3.1 Purpose of the Specification Language

Five main (not necessarily independent) criteria can be used to characterize the purpose of a specification language:

- First one can distinguish functional specification from specification which express the dynamics of the system which achieves a desired functionality.
- Secondly one can distinguish specification languages which aim at a mainly mathematical definition from specifications which use a specific conceptual model to specify a program.
- Thirdly one can ask whether a specification approach includes the notion of proof obligations which enforce the internal correctness of specifications.
- Fourthly one can ask whether a specification approach provides a refinement (also called reification) calculus which enforces the correct implementation of a specified system (i.e., which supports the transformation from specifications to implementations).
- Finally one can ask whether specification approaches aim at prototyping as a means to evaluate specifications.

The characterization of the languages according to these criteria is provided by table 2.

Table 2. Purpose of the languages

Language	(1) Specification type	(2) Coupled with a conceptual model	(3) Proof obligations	(4) Refinement calculus	(5) Prototyping
KE					
DESIRE	dynamics	yes	no	no	yes
KARL	dynamics	yes	no	no	yes
$K_{BS}SF$	dynamics	yes	no	no	yes
$(ML)^2$	dynamics	yes	no	no	no
TFL	dynamics	yes	no	no	no
SE					
ADT	functional	no	no/yes[a]	yes	partial
DEL/CMSL	dynamics	yes	no	no	no
Ealgebras	dynamics	no	no	no (yes)[b]	yes
MLCM/COLD	dynamics	yes	no	no	no
VDM	functional & dynamics	no (yes)[c]	yes	yes	partial
Z	functional	no (yes)[c]	yes	no[d]	no

a. There exist algebraic approaches which include a notion of proof obligations.
b. There are rich practice and theory of refining evolving algebras but no explicit calculus.
c. Originally, a conceptual model had not been provided but recent development have defined derivates which are coupled with a conceptual model.
d. Wood (1993) discusses the use of refinement calculi for Z.

Interesting to note is the uniformity of the KE languages according to the first four criteria. Only the fifth criterion does not deliver a clear picture for these languages. In the SE domain a broader variation of properties can be found but it should be noted that more recent

27. A third aspect for the comparison which will not be discussed is the formal semantics which is used by the languages. In particular, the semantics of a state and state transitions are interesting in this context. Languages such as PDDL (Spruit et al., 1995), DDL (Spruit et al., 1993), and Transaction Logic (Bonner & Kifer, 1993), which were developed for specifying database updates have very interesting features compared with knowledge specification languages. They provide a complex notion for a state (which is described by a database) and powerful transitions between states (database updates) and their control. They thus resemble most of the needs which are required to express the dynamic reasoning process of knowledge-based systems.

approaches allows the specification of the internal dynamics of the systems whereas more traditional approaches aim on a pure functional specification.

Each of the approaches in KE that have been mentioned combines both types of description (i.e., functional and non-functional specifications). They specify control over functionally specified substeps. None of them provide a pure functional specification of the entire system. This reflects the fact that human expertise required for kbs is mostly given in an operational manner. Expertise is provided as knowing *how* to do things (Berry, 1987; Schreiber & Birmingham (eds.), 1995). Normally, experts do not have an explicit and declarative description of the functionality of their knowledge. As a consequence, *the relationship between pure functional specifications and specifications making commitments on the algorithmic realization of the functionality is the reverse in knowledge engineering as compared to software engineering*. For example, in VDM a functional specification of an operation by a pair of pre- and post-conditions is the starting point which becomes *refined* (reified) during design. In knowledge engineering, the functionality of a kbs is always specified in an operational manner by a procedure which realises the desired functionality.[28] It would therefore require an *abstraction* step to derive a pure functional specification from it. Future work will probably try to develop such reverse abstraction calculi and to integrate a pure functional description level of the overall system into the knowledge specification languages.

3.2 Knowledge Level Specifications?

A *conceptual model* underlying a language provides a strong *bias* on how things become expressed in it. It improves the understandability of a formal specification by linking it to the informal specification. We selected five features of conceptual models for specifying kbs to compare the differences of the approaches at a conceptual level.

- (1) We ask whether there are rich modelling primitives to express terminological knowledge (mostly suited at a domain layer in KADS model of expertise).
- (2) we ask for the generic specification of inferences and for the (3) object-meta distinction between the domain and the inference layers.
- The two remaining criteria concern the representation of dynamics: (4) the representation of a state, and (5) the definition of control over the execution of these elementary transitions.

The characterization of the languages according to these criteria is provided by table 3.

Table 3. Could a language appropriately specify the different aspects of a conceptual model of a kbs

Language	(1) Terminological knowledge	(2) Generic inferences	(3) Object/ meta level	(4) Notion of states	(5) Control
KE					
DESIRE	poor	yes	yes	termination states of modules and the truth values of their atoms	constructive by rules
KARL	rich	yes	limited	a fixed set of dynamic variables containing sets of grounds facts	constructive procedural

28. Only its substeps, which are regarded to be atomic at the level of specification, are described purely functionally.

Table 3. Could a language appropriately specify the different aspects of a conceptual model of a kbs

Language	(1) Terminological knowledge	(2) Generic inferences	(3) Object/ meta level	(4) Notion of states	(5) Control
$K_{BS}SF$	poor	yes	no	a fixed set of dynamic variables containing sets of formulas	constructive procedural
$(ML)^2$	poor	yes	yes	a fixed set of dynamic variables containing lists of ground facts	constructive procedural
TFL	poor	yes	no	no	constructive by rules and procedural
SE					
ADT	poor	yes	no	no	---
DEL/CMSL	rich	yes	no	attribute values of objects	non-constructive by constraints
Ealgebras	poor	yes	no	algebra	constructive by rules
MLCM/COLD	rich	yes	no	algebra	constructive procedural
VDM	poor	yes	no	a record of dynamic variables + invariants	constructive procedural
Z	poor	yes	no	state schemas	---

Terminological Knowledge: Most of the languages provide constants, sorts/types, functions, predicates/relations, and some mathematical toolkit as a means to specify the static aspects of a system. We classify such languages as poor. Rich languages provide additional syntactical sugar on top of these mathematical primitives. In particular, specification languages for information systems (originally for databases) provide a rich variety of appropriate modelling primitives for expressing static system aspects: values, objects, classes, attributes with domain and range restrictions, set-valued attributes, is-a relationships with attribute inheritance, aggregation, grouping etc. (see Brodie, 1984; Brodie & Ridjanovic, 1984; Wieringa, 1991b; Jungclaus, 1993; Kifer et al.,1993). In the case of KARL it was the application of these ideas which provide modelling primitives at a high conceptual level. This enables a smooth transformation from semiformal specification languages like CML (Schreiber et al., 1993) for the domain layer of the KADS model of expertise to formal specifications of these models (Fensel & Neubert, 1994).

Generic Inferences and Object-Meta Level Relationship: Whereas the requirement for generic specifications do not have discrimination power between the languages this is quite different for the object/meta-level relationships. No specification language from SE and even not all of the specification languages from KE provide such a relationship where predicates from one logical language are used as terms by a second logical language. Such a relationship is appropriate to express the control of the reasoning process of the object-level language by a meta-level language.

Notion of Dynamics: The notion of dynamics implies the characterisation of states and state transitions. Table 3 shows a broad variation of syntactical representations of a state. The KADS-oriented languages KARL and $(ML)^2$ represent a state by the contents of knowledge roles. In KARL only the current state is stored as contents of a knowledge role whereas in $(ML)^2$ the *history* of local states is stored in a knowledge role. The representation of the history is only locally available for each role. That is, the global history of the reasoning process is not represented as past states of different roles are not related in any ordering.

Technically these knowledge roles are modelled by variables of dynamic logic (Harel, 1984). A state in dynamic logic is characterized by the value assignments of each dynamic variable. A richer characterization of a state—compared to dynamic logic—is provided by the algebraic approaches evolving algebras and COLD which characterize a state by an algebra. Dynamic logic provides a flat list of dynamic variables to represent a state whereas an algebra with its internal structure provides a much richer data structure to express a state. The static characterization of a state by *invariants* as provided by VDM-SL and Z becomes very useful when one wants to specify conditions for the correctness of the reasoning process. Still, these invariants in VDM and Z cannot use the history of the reasoning process for defining constraints on legal states.

With regard to the last criterion one can distinguish non-constructive and constructive means to specify control over state transitions. A *non-constructive specification of control* defines *constraints* for legal control flows. That is, they exclude possible control flows but do not define actual ones. Examples for such a specification can be found in the domain of information system specifications, e.g., dynamic equational logic (DEL) and TROLL (Jungclaus, 1993). *Constructive specifications* of control flow define directly the actual control flow of a system and apply a variant of the closed-world assumption. That is, each control flow which is not defined is not possible. Such definitions of control can in principle be used to simulate the behavior of a system. In general, there is no clear cut line between both approaches as constructive definitions of control could allow non-determinism which again leads to several possibilities for the actual control. Again, two variants can be distinguished for the constructive definition of control: a language like $(ML)^2$ defines the control flow globally by a *procedural* language. Approaches such as DESIRE and evolving algebras define the control flow locally by *rules*. Each rule defines a precondition and an operation which could be executed if the precondition was evaluated to true. The overall control flow is defined by a set of rules and a rule interpreter. TSL combines both types of constructive definition of control flow in order to allow the procedural definition of control as well as strategic reasoning over different alternatives for control. This corresponds to DESIRE which aims not only at the specification of control but also at the specification of strategic reasoning over control.

4 Conclusion

Software engineering provides an impressive arsenal of formal specification languages and techniques to verify these specifications. Refinement or reification calculi stress the development of correct implementations from formal specifications. The situation in knowledge engineering is quite different. Early specification languages for kbs such as OMOS (Linster, 1992) or MODEL-K (Karbach & Voss, 1993) did not at all aim at formalization, but provided executable languages to support knowledge acquisition by prototyping. Meanwhile, a number of languages with a solid formal semantics have been developed, but still none of them provides proof calculi which support the verification of formally specified properties of specifications. Two reasons could be responsible for this. First, software engineering started developing formal specification methods 25 years ago.[29] The development of formal specification methods for kbs is a more recent phenomenon. Second, knowledge specification languages should not only specify the functionality of kbs but also heuristic knowledge to achieve the functionality in an efficient manner. Often, this heuristic knowledge exists only implicitly as human expertise. Large parts of this expertise are based on tacit knowledge (Berry, 1987). Knowledge acquisition is a modelling activity which tries to come up with an explicit model of this knowledge. There is a high need for tools which support this modelling process by evaluating these models. Prototyping is a very useful tool for this purpose. The development of formally correct specifications and implementations is of less concern during this activity. An exception is the work of Aben (1995) who provides a formal calculus based on a pre- and post-condition notions for inference actions.

A further difference, which probably stems from the same reason, is that knowledge specification languages make much more use of semiformal specification techniques than comparable techniques in software engineering. Actually, most of the knowledge specification languages were developed to formally refine semiformal specification techniques like the KADS model of expertise. Therefore, these languages provide strong support during the conceptual modelling process, and formal specifications are structured using different modelling primitives and knowledge types stemming from semiformal techniques. The understandability of formal specifications is improved and the process of writing formal specifications is simplified. On the one hand, the use of semiformal specification techniques can also be found in software engineering, especially by specification languages which are designed for the early phase during software development (called requirements engineering) or as a result of the current trend of object-oriented specification techniques. On the other hand, the use of modelling primitives at a high conceptual level for kbs is not just a question of emphasis but it is enabled by introducing assumptions about the type of system which can be meaningfully specified with such a language. Specification languages for kbs are special-purpose languages which provide support at a higher level by restricting their scope of applicability.[30]

Finally, a difference between knowledge specification languages and specification languages in Software Engineering lies in the fact that most of the latter aim for a declarative specification of the functionality of a system. They try to abstract from *how* this functionality is achieved. As already mentioned, in knowledge-based system development part of the "*how*" is regarded as essential expertise which must be specified. Therefore, an inference layer in $(ML)^2$ or KARL specifies the significant inferences of a problem-solving process and the task layer supplements these definitions with a control flow which should ensure an effective and efficient computation of a solution. They specify in an abstract manner (i.e., by defining control over functionally specified substeps) how a solution is achieved instead of only describing *what* the solution is. This circumstance has serious implications for semantics and proof calculi of these languages as both have to include functional and non-functional (i.e., dynamic) aspects. This character of knowledge specification languages also relates them to approaches which are provided for the *design phase* of software systems as the procedural constructs in VDM or the algorithmic description style of evolving algebras.

---

29. Early approaches in software engineering also provided executable specification languages without trying to support formalization, e.g., RSL (cf. Bell et al., 1977; Alford, 1990), Gist (cf. Balzer et al., 1982) or PAISLey (Zave, 1982 & 1991).

30. The same can also be seen in the domain of specification languages for information systems which use the restricted class of software artefacts as strong bias in providing language primitives at a higher conceptual level.

Acknowledgements

I would like to thank Ernst-Erich Doberkat, Pascal van Eck, Joeri Engelfriet, Marie-Claude Gaudel, Rix Groenboom, Yuri Gurevich, Frank van Harmelen, Thorsten Hartmann, Gerard R. Renardel de Lavalette, Paul Krause, Andreas Oberweis, Claus Pahl, Joachim Posegga, Gunter Saake, Remco Straatman, Rudi Studer, Jan Treur, Niek Wijngaards, Mark Willems, and two anonymous reviewers for very helpful comments on drafts of the paper.

References

Aben, M. (1995): *Formal Methods in Knowledge Engineering*, PhD dissertation, University of Amsterdam.

Alford, M. (1990): SREM at the Age of Eight; the Distributed Computing Design System. In R. H. Thayer et al. (eds.), *System and Software Requirements Engineering*, IEEE Computer Society Press, Washington.

Andrews, D. and Ince, D. (1991): *Practical Formal Methods with VDM*, Mc Graw Hill.

Andersen, M., Elmstr∅m, R., Lassen, P. B. and Larsen, P. G. (1992): Making Specifications Executable—Using IPTES Meta-IV, *Microprocessing and Microprogramming*, vol 35.

Angele, J. (1993): *Operationalisierung des Modells der Expertise mit KARL*, Infix, St. Augustin.

Angele, J., Fensel, D., Landes, D., Neubert, S. and Studer, R. (1993): Model-based and Incremental Knowledge Engineering: The MIKE Approach. In J. Cuena (ed.), *Knowledge Oriented Software Design*, IFIP Transactions A-27, Elsevier Science Publisher B.V., Amsterdam.

Angele, J., Fensel, D. and R. Studer, R. (1994): The Model of Expertise in KARL. In *Proceedings of the 2nd World Congress on Expert Systems*, Lisbon/Estoril, Portugal, January 10-14.

Baeten, J. C. M. (1990): *Applications of Process Algebra*, Cambridge Tracts in Theoretical Computer Science, no 17, Cambridge University Press, Cambridge.

Baeten, J. C. M. and Weijland, W. P. (1990): *Process Algebra*, Cambridge Tracts in Theoretical Computer Science, no 18, Cambridge University Press, Cambridge.

Balzer, R. and Goldman, N. (1979): Principles of Good Software Specification and their Implications for Specification Language. In *Proceedings of Reliable Software (SRS)*, Boston, Massachusetts.

Balzer, R. M., Goldman, N. M. and Wile, D. S. (1982): Operational Specifications as the Basis for Rapid Prototyping, *ACM SIGSOFT Software Engineering Notes*, 7(5), December 1982.

Bauer, F. L., Ehler, H., Horsch, A., Möller, B., Partsch, H., Paukner, O. and Pepper, P. (1987): *The Munich Project CIP, vol II: The Program Transformation System CIP-S*, Lecture Notes on Computer Science (LNCS), no 292, Springer-Verlag, Berlin.

Beckert, B. and Posegga, J. (1995): lean*E*A: A Poor Man's Evolving Algebra Compiler, research report, University of Karlsruhe, Fakultaet fuer Informatik.

Bell, T. E., Bixler, D. C. and Dyer, M. E. (1977): An Extendable Approach to Computer-Aided Software Requirements Engineering, *IEEE Transactions on Software Engineering*, 3(1).

Berry, D. C. (1987): *The Problem of Implicit Knowledge*, Expert Systems, 4(3).

Bicarregui, J. C., Fitzgerald, J. S., Lindsay, P. A., Moore, R., and Ritchie, B. (1994): *Proof in VDM: A Practitioner´s Guide*, Springer Verlag, Berlin.

Bidoit, M. (1989): *PLUSS, un langage pour le developpement de specifications algebriques modulaires*, These d'Etat, University Paris Sud, Orsay.

Bidoit, M., Kreowski, H.-J., Lescane, P., Orejas, F. and Sannella, D. (eds.) (1991): *Algebraic System Specification and Development*, Lecture Notes in Computer Science, no 501, Springer-Verlag, Berlin.

Bjorner, D., Hoare, C. A. R. and Langmaack, H. (eds.) (1990): *VDM´90. VDM and Z - Formal Methods in Software Development*, Lecture Notes in Computer Science, no 428, Springer-Verlag, Berlin.

E. Börger, Gurevich, Y. and Rosenzweig, D. (1994): The Bakery Algorithm: Yet Another Specification and Verification. In E.B. Börger (ed.), *Specification and Validation Methods*, Oxford University Press.

Brazier, F. M. T., Keplicz, B. D., Jennings, N. R. and Treur, J. (1995a): Formal Specifications of Multi-Agent Systems: A Real-World Case. In *Proceedings of the 1st International Conference on Multi-Agent Systems (ICMAS-95)*, San Fransisco, CA, June 12-14.

Brazier, F. M. T. and Treur, J. (1994): User Centered Knowledge-Based System Design: A Formal Modelling Approach. In Steels, L. et al. (eds.) *A Future for Knowledge Acquisition, 8th European Knowledge Acquisition Workshop, EKAW-94,* Lecture Notes in Artificial Intelligence, no 867, Springer-Verlag.

Brazier, F. M. T., Treur, J., Wijngaards, N. J. E. and Willems, M. (1995b): Formal Specification of Hierarchically (De)Composed Tasks. In *Proceedings of the 9th Banff Knowledge Acquisition For Knowledge-Based Systems Workshops (KAW-95)*, Banff, Alberta, Canada, February 26 - March 3.

Breuker, J. A. and van de Velde, W. (eds.) (1994): *The CommonKADS Library For Expertise Modelling*, IOS Press, Amsterdam.

Bonner, A. J. and Kifer, M. (1993): Transaction Logic Programming. In *Proceedings of the 10th International Conference on Logic Programming (ICLP)*, Budapest, Hungary, June 21-24.

Bourdeau, R. H. and Cheng B. H. C. (1995): A Formal Semantics for Object Model Diagrams, *IEEE Transactions on Software Engineering,* 21(10).

Brodie, M. L. (1984): On the development of data models. In Brodie et al. (eds.), *On Conceptual Modeling*, Springer-Verlag, Berlin.

Brodie, M. L. and Ridjanovic, D. (1984). On the design and specification of database transactions. In Brodie et al. (eds.), *On Conceptual Modeling*, Springer-Verlag, Berlin.

Brooking, A. G. (1986): The Analysis Phase in Development of Knowledge-Based Systems. In W. A. Gale (ed.), *AI and Statistic*, Addison-Wesley Publishing Company, Reading, Massachusetts.

Chandrasekaran, B. (1986): Generic Tasks in Knowledge-Based Reasoning: High-Level Building Blocks for Expert System Design, *IEEE Expert*, 1(3).

Clancey, W. J. (1987): From Guiden to Neomycin and Heracles in Twenty Short Lessons. In A. van Lamsweerde, ed., *Current Issues in Expert Systems,* Academic Press.

Coad, P. and Yourdon, E. (1991): *Object-Oriented Analysis*, 2nd ed., Yourdon Press, Englewood Cliffs.

Craigen, D., Gerhart, S. and Ralston, T. (1993): *An International Survey of Industrial Applications of Formal Methods*, vol 1 and 2, U.S. Department of Commerce, National Institute of Standards and Technology, Gaithersburg, report NISTGCR 93/626. ftp nemo.ncsl.nist.gov, pub/ahis/formal_methods. A short version is appeared as Gerhart, S. , Craigen, D. and Ralston,T. (1993): Observations on Industrial Practice Using Formal Methods. In *Proceedings of the 15th International Conference on Software Engineering (ICSE-93)*, May 17-21, Baltimore, Maryland.

Doberkat, E.-E. (1994): *Generating an Algebraic Specification from an ER-Model*, STW Memo, no ISSN 0933-7725, Fachbereich Informatik, Univerity of Dortmund, Germany.

Doberkat, E.-E. and Fox, D. (1989): *Software Prototyping mit SETL*, Leitfäden und Monographien der Informatik, Teubner-Verlag, Stuttgart.

Dorfman, M. (1990): System and Software Requierements Engineering. In R. H. Thayer and M. Dorfman (eds.): *System and Software Requierements Engineering*, IEEE Computer Society Press, Los Alamitos, California.

Ehrig, H. and B. Mahr, B. (eds.) (1985): *Fundamentals of Algebraic Specifications 1*, Springer-Verlag, Berlin.

Ehrig, H. and B. Mahr, B. (eds.) (1990): *Fundamentals of Algebraic Specifications 2*, Springer-Verlag, Berlin.

Elmstr∅m, R., Lassen, P. B. and Larsen, P. G. (1994): The IFAD VDM-SL Toolbox: A Practical Approach to Formal Specifications. In *ACM SIGPLAN Notices.*

Elmstr∅m, R., Lintulampi, R. and Pezze, M. (1993): Giving Semantics to SA/RT by Means of High Level Timed Petri Nets, *Real-Time Systems*, vol 5, no 2-3.

Feijs, L. M. G. and Jonkers,H. B. M. (1992): *Formal Specification and Design*, Cambridge Tracts in Theoretical Computer Science, no 35.

Feijs, L.M.G. , Jonkers, H.B.M., Koymans, C.P.J. and Renardel de Lavalette, G.R. (1989): *Formal definition of the design language COLD-K* (Preliminary version)}, ESPRIT document METEOR/t7/PRLE/7, April 1987 (Final version: August 1989).

Fensel, D. (1995): *The Knowledge Acquisition and Representation Language KARL*, Kluwer Academic Publ., Boston.

Fensel, D., Angele, J., Landes, D. and Studer, R. (1993): Giving Structured Analysis Techniques A Formal And Operational Semantics With KARL. In H. Züllighoven et al. (eds.), *Requirements Engineering ´93: Prototyping,* Teubner Verlag, Stuttgart.

Fensel, D. and van Harmelen, F. (1994): A Comparison of Languages which Operationalize and Formalize

KADS Models of Expertise, *The Knowledge Engineering Review*, 9(2).

Fensel, D. and Neubert, S. (1994): Integration Of Semiformal and Formal Methods For Specification of Knowledge-Based Systems. In B. Wolfinger (ed.), *Innovation bei Rechen- und Kommunikationssystemen*, Informatik Aktuell, Springer-Verlag, Berlin.

France, R. B. and Docker, T. W. G. (1989): Formal Specifications Using Structured System Analysis. In *Proceedings of the 2nd European Software Engineering Conference ESEC´89*, Warwick, September 11-15, Lecture Notes in Computer Science, no 387, Springer-Verlag, Berlin.

Fuchs, N. E. (1992): Specifications Are (Preferably) Executable, *Software Engineering Journal*, 7.

Futatsugi, K., Goguen, J. A., Jouannaud, J.-P. and Meseguer, J. (1985): Principles of OBJ2. In *Proceedings of the 12th ACM Symposium on Principles of Programming Languages*, New Orleans.

Gaudel, M.-C. (1984): *A First Introduction to PLUSS*, technical report, LRI, Universite Paris Sud, Orsay.

Gaudel, M.-C. (1990): Algebraic Specifications. In J. McDermied (ed.), *Software Engineer's Reference*, Butterworths.

Gaudel, M.-C. (1994): Formal Specification Techniques. In *Proceedings of the 16th International Conference on Software Engineering (ICSE-94)*, May 16-21, Sorrento, Italy.

Gavrila, I. and Treur, J. (1993): A Formal Model for the Dynamics of Compositional Reasoning Systems. In *Proceedings of the 11th European Conference on Artificial Intelligence (ECAI-94)*, Amsterdam, The Netherlands, August 8-12. 1994. An extended version is available as research report, no IR-323, Faculteit der Wiskunde en Informatica, Vrije Univeriteit Amsterdam.

Geelen,P., Ruttkay, Z. and Treur, J. (1991): *Logical Analysis and Specification of an Office Assignment Task*, research report, no IR-283, Vrije Universiteit Amsterdam, Faculteit der Wiskunde en Informatica.

Gerrard, C. P., Coleman, D. and Gallimore, M. (1990): Formal Specification and Design Time Testing, *IEEE Transactions on Software Engineering*, 16(1).

Groenboom, R. and Renardel de Lavalette, G., R. (1994): Reasoning about Dynamic Features in Specification Languages. In D.J. Andrews et al. (eds.), *Semantics of Specification Languages*, Springer Verlag, Berlin.

Groenboom, R. and Renardel de Lavalette, G., R. (1995): A Formalisation for Evolving Algebra. In *Proceedings of Accolade 95, Dutch Graduate School in Logic*, Amsterdam.

Gurevich, Y. (1993): *Evolving Algebras. A Tutorial Introduction*. In E. G. Rozenberg et al. (eds.), Current Trends in Theoretical Computer Science, World Scientific.

Gurevich, Y. (1994): Evolving Algebras 1993: Lipari Guide. In E.B. Börger (ed.), *Specification and Validation Methods*, Oxford University Press.

Harel, D. (1984): Dynamic Logic. In D. Gabby et al. (eds.), *Handook of Philosophical Logic*, vol. II, *Extensions of Classical Logic*, Publishing Company, Dordrecht (NL).

van Harmelen, F. and J. Balder, J. (1992): $(ML)^2$: A Formal Language for KADS Conceptual Models, *Knowledge Acquisition*, 4(1).

van Harmelen, F. and Fensel, D. (1995): Formal Methods in Knowledge Engineering, *The Knowledge Engineering Review*, 10(4).

Hasselbring, W. (1994): Prototyping Parallel Algorithms in a Set-Oriented Language, Verlag Dr. Kovac, Hamburg, Germany.

Hayes, I. J. (1992): VDM and Z. A Comperative Case Study, *Formal Aspects of Computing*, 4(1).

Hayes, I. J. and Jones, C. B. (1989): Specifications Are Not (Necessarily) Executable, *Software Engineering Journal*, 4(6).

Hayes, I. J., Jones, C. B. and Nicholls, J. E. (1994): Understanding the Difference Between VDM and Z, *ACM Sigsoft Software Engineering Notes*, 19(3).

He, X. (1995): PZ Nets - A Formal Method Integrating Petri Nets with Z. In *Proceedings of the 7th International Conference on Software Engineering and Knowledge Engineering* (*SEKE-95*), Rockville, Maryland, June 22-24.

Huggins, J. K. (1993): *Evolving Algebra Interpreter*, Manuscript University of Michican.

Jones, C. B. (1990): *Systematic Software Development Using VDM*, 2nd ed., Prentice Hall.

Jungclaus, R. (1993): *Modeling of Dynamic Object Systems - A Logic-based Approach*, Vieweg Verlag.

Kappel, A. M. (1993): Executable Specifcations based on Dynamic Algebras. In *Proceedings of the 4th International Conference on Logic Programming and Automated Reasoning (LPAR-93)*, St. Petersburg, Russia, July 13-20.

Karbach, W. and Voß, A. (1993): MODEL-K for Prototyping and Strategic Reasoning at the Knowledge Level. In .-M. David et al. (eds.): *Second Generation Expert Systems*, Springer-Verlag, Berlin.

Kifer, M., Lausen, G. and Wu, J. (1993): *Logical Foundations of Object-Oriented and Frame-Based Languages*. In Technical Report 93/06, Department of Computer Science, SUNY at Stony Brook, NY. To appear in *Journal of the ACM*.

Kozen, D. (1990): Logics of Programs. In J. v. Leeuwen (ed.), *Handbook of Theoretical Computer Science*, Elsevier Science Publ., B. V., Amsterdam.

Krause, P., Fox, J., O`Neill, M. and Glowinski, A. (1993): Can We Formally Specify a Medical decision Support Systems, *IEEE Expert*, vol 8(3).

van Langevelde, I., Philipsen, A. and Treur, J. (1992): Formal Specification of Compositional Architectures. In *Proceedings of the 10th European Conference on Artificial Intelligence (ECAI-92)*, Vienna, Austria, August 3-7, 1992. An extended version is available as research report, no IR-282, Vrije Universiteit Amsterdam, Faculteit der Wiskunde en Informatica.

van Langevelde, I., Philipsen, A. and Treur, J. (1993): A Compositional Architecture for Simple Design Formally Specified in DESIRE. In J. Treur and Th. Wetter (eds.): *Formal Specification of Complex Reasoning Systems*, Ellis Horwood, New York.

Larsen, P. G., van Katwijk, J., Plat, N., Pronk, K. and Toetenel, H. (1991): SVDM: An Integrated Combination of SA and VDM. In *Proceedings of the Methods Integration Conference*, Leeds, UK.

Larsen, P. G., Plat, N. and Toetenel, H. (1993): A Formal Semantics of Data Flow Diagrams, *Formal Aspects of Computing*, vol 3.

Lassen, P. B. and Larsen, P. G. (1991): An Executable Subset of Meta-IV with Loose Specification. In *Proceedings of the VDM´91 Formal Software Development Methods, Noordwijkerhout*, The Netherlands, Oktober 1991, Springer-Verlag, Berlin.

Linster, M. (1992): *Knowledge Acquisition Based on Explicit Methods of Problem Solving*, PhD dissertation, University of Kaiserslautern.

Marcus, S. (ed.) (1988): *Automating Knowledge Acquisition for Experts Systems*, Kluwer Academic Publisher, Boston.

Mc Dermott, J. (1982): R1: A Rule-Based Configurer of Computer Systems, *Artificial Intelligence*, 19.

Milnes, B. G. (1992): *A Specification of the Soar Cognitive Architecture in Z*, research report CMU-CS-92-169, School of Computer Science, Carnegie Mellon University, Pittsburg, PA.

Nakagawa, A. T., Sakakihara, T. and Futatsugi, K. (1993): Algebraic Specification of Reasoning Systems. In J. Treur and Th. Wetter (eds.): *Formal Specification of Complex Reasoning Systems*, Ellis Horwood, New York.

Newell, A. (1982): The Knowledge Level, *Artificial Intelligence*, vol 18.

Oberweis, A., Scherrer, G., Stucky, W. (1994): INCOME/STAR: Methodology and Tools for the Development of Distributed Information Systems, *Information Systems*, 19(8).

Pierret-Golbreich, C. and Talon, X. (submitted 1995): An Algebraic Specification of the Dynamic Behaviour of Knowledge-Based Systems, *The Knowledge Engineering Review*.

Plotkin, G. D. (1981): *A Structural Approach to Operational Semantics*, technical report, no DAIMI FN-19, Aaarhus University, Denmark.

Przymusinski, T. C. (1988): On the Declarative Semantics of Deductive Databases and Logic Programs. In J. Minker (ed.), *Foundations of Deductive Databases and Logic Programming*, Morgan Kaufmann Publisher, Los Altos, CA.

Randell, G. P. (1990): *Translating Data Flow Diagrams into Z (and Vice Versa)*. Technical Report 90019, Procurement Executive, Ministry of Defence, RSRE, Malvern, Worcestershire, UK.

Ross, D. T. (1977): *Structured Analysis (SA): a Language for Communicating Ideas*, IEEE Transactions on Software Engineering, 3(1).

Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., and Lorensen, W. (1991): *Object-Oriented Modelling and Design*, Prentice Hall, Englewood Cliffs, New Jersey.

Ryan, P. and C. Sennett, C. (1993): *Formal Methods in Systems Engineering*, Springer Verlag, Berlin.

Schreiber, A. Th. (1992): *Pragmatics of the Knowledge Level*, PhD dissertation, University of Amsterdam.

Schreiber, A. Th. and Birmingham, B. (eds.) (1995): special issue on the *VT Sisyphus Task, International Journal* *of Human-Computer Studies* (*IJHCS*), special issue, in press.

Schreiber, A. TH., Wielinga, B. J., Akkermans, H., Van De Velde, W. and de Hoog, R. (1994): CommonKADS. A Comprehensive Methodology for KBS Development, *IEEE Expert*, 9(6).

Schreiber, A. TH., Wielinga, B. J., and Breuker, J. (eds.) (1993): *KADS. A Principled Approach to Knowledge-Based System Development*, Academic Press, London.

Schreiber, A. Th., Wielinga, B. J. and Jansweijer, W. (1995): The KACTUS View on the ‘ O’ Word. In *Proceedings of the Dutch National Conference on AI (NAIC-95)*, Rotterdam.

Schwarz, J. T., Dewar, R., Dubinski, E. and Schonberg, E. (1986): *Programming with Sets: An Introduction to SETL*, Springer-Verlag, Berlin.

Semmens, L. T., France, R. B., and Docker, T. W. G. (1992): Integrating Structured Analysis and Formal Specification Techniques, *The Computer Journal,* 35(6).

Sernadas, A., Sernadas, C. and Costa, J.F. (1992): *Object Specification Logic*. Research Report INESC/DMIST, University of Lisbon. To appear in *Journal of Logic and Computation*.

Sheppard, D. (1995): *An Introduction to Formal Specification with Z and VDM*, McGraw-Hill, London.

Spee, J. W. and in ‘t Veld, L. (1994): The Semantics of K_{BS}SF: A Language For KBS Design, *Knowledge Acquisition*, vol 6.

Spivey, J. M. (1988): *Understanding Z. A Specification Language and Its Formal Semantics*, Cambridge University Press, Cambridge.

Spivey, J. M. (1992): *The Z Notation. A Reference Manual*, 2nd ed., Prentice Hall, New York.

Spruit, P., A., Wieringa, R., J., and Meyer, J.-J., C. (1993): Dynamic Database Logic: the First-order Case. In V. W. Lipeck et al. (eds.), *Modelling Database Dynamics*, Springer-Verlag, Berlin.

Spruit, P., A., Wieringa, R. , J., and Meyer, J.-J., C. (1995): Axiomatization, Declarative Semantics and Operational Semantics of Passive and Active Updates in Logic Databases, *Journal of Logic Computation*, 5(1).

Stepney, S., Barden, R. and Cooper, D. (eds.) (1992): *Object Orientation in Z*, Springer-Verlag, Berlin.

Treur, J. (1992): *Interaction Types and Chemistry of Generic Task Models*. In M. Linster et al. (eds.), *Proceedings of the European Knowledge Acquisition Workshop (EKAW-91)*, GMD Studien, no 211.

Treur, J. (1994): Temporal Semantics of Meta-Level Architectures for Dynamic Control of Reasoning. In L. Fribourg et al. (eds.), *Logic Program Synthesis and Transformation - Meta Programming in Logic, Proceedings of the 4th International Workshops, LOPSTER-94 and META-94*, Pisa, Italy, June 20-21, 1994, Lecture Notes in Computer Science, no 883, Springer Verlag-Berlin.

Treur, J. and Wetter, Th. (eds.) (1993): *Formal Specification of Complex Reasoning Systems*, Ellis Horwood, New York.

Turner, J., G. and McCluskey, T., L. (1994): *The Construction of Formal Specifications*, McGraw-Hill, London, 1994.

in ‘t Veld, L., Jonker, W. and Spee, J. W. (1993): The Specification of Complex Reasoning Tasks in KBSSF. In J. Treur and Th. Wetter (eds.): *Formal Specification of Complex Reasoning Systems*, Ellis Horwood, New York.

Voss, H. and Voss, A. (1993): Reuse-Oriented Knowledge Engineering with MoMo. In *Proceedings of the 5th International Conference on Software Engineering and Knowledge Engineering (SEKE93)*, San Fransisco Bay, June 14-18.

Wielinga, B. J. and Schreiber, A. Th. (1994): Conceptual Modeling Of Large Reusable Knowledge Bases. In K. von Luck et al. (eds.), *Management and Processing of Complex Data Structures*, Lecture Notes in Computer Science, no 777, Springer-Verlag, Berlin.

Wielinga, B. J. Schreiber, A. Th. and Breuker, J. A. (1992): KADS: A Modelling Approach to Knowledge Engineering, *Knowledge Acquisition*, 4(1).

Wieringa, R. J. (1991a): A Formalization Of Objects Using Equational Dynamic Logic. In *Proceedings of the 2nd International Conference on Deductive and Object-Oriented Databases (DOOD-91)*, Munich, Germany, December 16-18, 1991, C. Delobel et al. (eds.), Springer-Verlag, Berlin.

Wieringa, R. J. (1991b): Steps towards a method for the formal modeling of dynamic objects, *Data and Knowledge Engineering,* vol 6.

Wieringa, R. J. and van de Riet, R. P. (1990): Algebraic Specification of Object Dynamics in Knowledge Based Domains. In R. A. Meersman et al. (eds.), *Artificial Intelligence, Databases and Information Systems* (DS-

3), North-Holland.

Wirsing, M. (1990): Algebraic Specification. In J. van Leeuwen (ed.), *Handbook of Theoretical Computer Science*, Elsevier Science Publ.

Wood, K. R. (1993): A Practical Approach to Software Engineering Using Z and The Refinement Calculus. In *Proceedings of the First ACM SIGSOFT Symposium on the Foundations of Software Engineerin*g, Los Angeles, CA, December 7-10, 1993, *ACM Software Engineering Notes*, vol 18(5).

Woodcock, J. C. P. and Larsen, P. G. (eds.) (1993): FME´93: Industrial-Strength Formal Methods, *Proceedings of the First International Symposium of Formal Methods Europe*, Odense, Denmark, April 19-23, 1993, Lecture Notes in Computer Science, no 670, Springer-Verlag, Berlin.

Wordsworth, J. B. (1992): *Software Development with Z*, Addison-Wesley.

Yourdon, E. (1989): *Modern Structured Analysis*, Prentice-Hall, Englewood Cliffs.

Zave, P. (1982): An Operational Approach to Requirements Specification for Embedded Systems, *IEEE Transactions on Software Engineering*, 3(8).

Zave, P. (1991): An Insider´s Evaluation of PAISLey, *IEEE Transactions on Software Engineering*, 17(3).