Data and Knowledge Engineering, 26(3):243-269, 1998

Modal Change Logic (MCL): Specifying the Reasoning of Knowledge-based Systems

Dieter Fensel[1], Rix Groenboom[2] and G. R. Renardel de Lavalette[2]

[1] University of Karlsruhe, Institute AIFB, D-76128 Karlsruhe, Germany. E-mail: dieter.fensel@aifb.uni-karlsruhe.de

[2] University of Groningen, Department of Computing Science, P.O. Box 800, 9700 AV Groningen, the Netherlands. E-mail: {rix | grl}@cs.rug.nl

Abstract. We investigate the formal specification of the *reasoning process* of knowledge-based systems in this paper. We analyze the corresponding parts of the KADS specification languages KARL and $(ML)^2$ and deduce some general requirements. The essence of these languages is that they integrate a declarative specification of inferences with control information. The languages differ in the way they achieve this integration and each of them has shortcomings. We propose a unifying semantical framework that integrates the core of the different solutions and overcomes their problems. We define a semantics and axiomatization with the *Modal Change Logic (MCL)*. The main contribution of the paper is *not* to introduce yet another specification language. Instead we aim at four goals: (1) defining a framework for describing the dynamic reasoning behavior of knowledge-based systems which integrates existing approaches; (2) defining a semantics for the specification of the dynamic reasoning behavior of a knowledge-based system within the *states as algebras* setting that overcomes several shortcomings and ad-hoc solutions of existing approaches; and (3) providing an axiomatization that enables the development of mechanized proof support. (4) Through conceptual and semantical clarity, we investigate the relationships to similar work in software engineering and database engineering opening possibilities for further cross-fertilization of these fields.

1 Introduction

The *model of expertise* as developed in the KADS-I [Schreiber et al., 1993] and CommonKADS projects [Schreiber et al., 1994] has become a widely used framework for developing and describing knowledge-based systems (KBSs). Such a model of expertise can be used to describe the reasoning process and the knowledge required by this process in an implementation-independent manner. During the last years a couple of formal or executable specification languages have been developed for describing KBSs. Most of them are based on the KADS model of expertise or define their conceptual model as a modification of this model. A survey of these languages can be found in [Treur & Wetter, 1993], [Fensel & van Harmelen, 1994], [Fensel, 1995c]. Supplementing conceptual modelling techniques like the

KADS model by formal specification languages has three well known advantages:

- Formal specification languages can be used to resolve ambiguity and missing details of specifications stated in natural language.
- Executable specification languages enable the evaluation of the specification by prototyping (i.e. testing).
- Proof calculi of the languages can be used to check relevant properties of a specification.

Common to all formal specification approaches for KBSs is that a formal semantics has to cover three aspects: the specification of static aspects of a KBS, the specification of the dynamics of a KBS (i.e., its reasoning), and the combination of both, i.e. its overall semantics. For our study we restrict our attention to the second and third parts, because we think that the main improvements are necessary in the dynamic part. This part also introduces the main distinction from many specification languages of software engineering which aim only at a pure functional description of a software system (cf. [Fensel, 1995c]). In general, most problems tackled with KBSs are inherently complex and intractable ([Bylander, 1991], [Bylander et al., 1991], [Nebel, 1996], [Fensel & Straatman, 1996]). Besides a precise functional specification (i.e., the definitions of the goals that should be achieved by the KBS) it is therefore necessary to specify the reasoning process and its use of knowledge which enable reasonable problem-solving for the expected cases. An important part of the knowledge that must be specified is therefore knowledge about the way to achieve a solution and not just declarative knowledge about what a solution should be.

In this paper, we will discuss a semantic framework for specifying the dynamic reasoning process of a KBS. We start with an analysis of the existing approaches to come up with a framework that integrates these approaches. In fact, we take an analysis of the two KADS-languages KARL [Fensel, Angele & Studer] and $(ML)^2$ [van Harmelen & Balder, 1992] as our point of departure. The language $(ML)^2$ describes the reasoning behavior by combining first-order logic, meta-logic and quantified dynamic logic [Harel, 1984]. The language KARL was developed as part of the MIKE project [Angele et al., 1993] and provides a variant of Horn clause logic and a restricted version of dynamic logic for this purpose. We have chosen KARL and $(ML)^2$ for our exercise as both languages rely on dynamic logic to represent the dynamics of the reasoning process. As the technical core of the semantics of $K_{BS}SF$ [Spee & in 't Veld, 1994] is close to that of KARL, most of the results of the paper can also be applied to it. The other specification languages for KBSs use different means for specifying the dynamics of a KBS: Petri nets (MoMo [Voss & Voss, 1993]), process algebra (TFL [Pierret-Golbreich & Talon, 1996]), or temporal logic with linear time (DESIRE [Treur, 1994]). We will discuss some of them in the comparison section.

A serious shortcoming of all of the specification languages for KBSs appears when taking a closer look at the third advantage of formal specification languages: Proof calculi can be used to formally prove properties of specifications. Up to now, none of these languages provides such a support. We make a step in this direction by providing an axiomatization for our approach.

We achieve this by reusing work done in software engineering. Our starting point in software engineering is the wide-spectrum specification language COLD, which can be used to specify static and dynamic aspects of a system. COLD, Common Object-oriented Language for Design, was developed at Philips Research Eindhoven in several ESPRIT-projects. The main ideas for the language originated from Hans Jonkers. The formal definition of COLD-K and its semantics was given in 1987 in [Feijs et al., 1987]. The semantics is based on the many-sorted partial infinitary logic MPL_{ω}, see [Koymans & Renardel de Lavalette, 1989]. The textbook [Feijs & Jonkers, 1992] gives a good introduction to COLD-K, the kernel language of COLD. For the concept of state, COLD and the specification formalism Evolving Algebras[1] [Gurevich, 1994] use what we could call the *states as algebras* approach. In this approach, a state is modelled by a (many-sorted) algebra. State transitions are performed by procedures, and consist of the modification of functions and predicates and the creation of new objects. To reason about these kinds of formalisms a variant of Dynamic Logic [Harel, 1984] was studied. This resulted in the Modal logic of Creation and Modification (MLCM) [Groenboom & Renardel de Lavalette, 1994], a multi-modal logic for reasoning about state modifications. In this paper, we discuss MCL (Modal Change Logic), which generalizes MLCM. Basically, we introduce new elementary state transition types in MLCM which cover the grainsize of state transitions in knowledge-based reasoning. A predecessor of MCL is MLPM (Modal Logic of Predicate Modification) [Fensel & Groenboom, 1996], which introduces some of these new state transition into a subset of MLCM. However, MCL generalizes these transition types and integrates them in the general framework of MLCM. As a consequence, we get an approach that integrates existing proposals, that overcomes several of their shortcomings and ad-hoc solutions, that provides an axiomatization and enables the development of mechanized proof support. Finally we get a scalable approach that covers existing approaches of knowledge engineering and other areas of system specification, opening possibilities for further cross-fertilization of these fields.

The structure of this paper is as follows. In Section 2, we introduce the knowledge specification languages KARL and $(ML)^2$ focusing on their dynamics. We use the experience with these languages to derive requirements for an appropriate semantic framework for the specifications of the dynamics of the reasoning of KBSs. We discuss the logic MCL in Section 3. We provide syntax, semantics and an axiomatization. In Section 4, we use MCL to formalize the inference and control constructs of the KADS languages MLPM, $(ML)^2$ and KARL; the Evolving Algebras approach [Gurevich, 1994] of software engineering and the languages PDDL [Spruit et al., 1995] and DDL [Spruit et al., 1993] for specifying database updates. Finally, in Section 5 we provide a comparison with work that uses different solutions and we outline directions for future research in Section 6.

2 Specification Languages for Knowledge-based Systems

This section introduces the formal languages that form the basis of the paper. We sketch the KADS model of expertise and the two languages KARL and $(ML)^2$.

2.1 The Model of Expertise

The KADS languages KARL and $(ML)^2$ use variants of the KADS model of expertise as their conceptual framework for specifying a KBS. We will use a simple diagnostic task as an example to illustrate this model (see Figure 1). The task of the KBS consists of finding the

1. Now called Abstract State Machines (ASMs).

diagnosis with the highest preference for a given set of symptoms.

The task layer introduces the goal that should be achieved by the system and it decomposes the overall task into subtasks and defines control over them. It combines a functional specification with the specification of the dynamic reasoning process that realizes the functionality. The inference layer specifies the inference process that realizes the subtasks of the task layer. In our example it consists of two inference actions:

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

The knowledge role *finding* provides input to the inference action *generate*, the knowledge role *hypothesis* delivers the results of the reasoning of *generate* to *select*, and the knowledge role *diagnosis* provides the results of *select* as output.[2] The two knowledge roles *causality* and *preference* provide knowledge necessary for the inference process. It is mapped from the domain layer, which provides causal knowledge which can be used to relate findings to diagnoses and knowledge which can be used to assign preferences to possible diagnoses. A simple control flow at the task layer is defined by first executing *generate* and applying *select* to its output.

The model of expertise separates domain knowledge and control knowledge. The domain layer contains the static knowledge from the application domain and its terminology. The inference layer and the body of the task layer describe the dynamics 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 provides the definition of control over the execution of the inference steps.[3] This distinction between the domain-specific knowledge at the domain layer and the domain-independent description of the reasoning process at the inference and task layers enables the *reuse of domain knowledge* for different task and reasoning strategies and the reuse of reasoning strategies (called *problem-solving methods* [Schreiber et al., 1994]) in different domains.

If formal specification languages are not used, 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 some of these elements. As a consequence of our focus on dynamics we abstract from some aspects of the languages. In the case of KARL we abstract from all syntactical extensions of Horn logic by semantical data modelling primitives, and in the case of $(ML)^2$ we abstract from the object-meta relationship between the domain and the inference layers. These abstractions help to focus on the relevant aspects and simplify the formal parts of this paper.

2.2 $(ML)^2$

The sub-language of $(ML)^2$ [van Harmelen & Balder, 1992] used to model a domain layer is

---

2. Following the naming convention for types and sorts we give *singular* names to knowledge roles independent of whether they contain one or several elements.
3. The inference layer defines the data flow between inferences but not the order in which they are executed.

order-sorted first-order logic extended by modularization. *Instances* are modelled by constants, and sorts can be used to model *concepts*. Sorts and therefore concepts can be arranged in an *is-a hierarchy*. *Relationships* between instances of concepts are modelled by predicates of the according sorts. *Attributes* of instances of concepts are modelled by functions. Arbitrary first-order theories can be used to further specify the defined 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 consists of axioms (i.e., logical formulae). These modules, i.e. sub-theories, can be combined by a union operator.

Each inference action (called ***p**rimitive **i**nference **a**ction* in (ML)^2) is modelled by a predicate and a theory which further specifies this predicate. In our running example, the inference actions *generate* and *select* are modelled by two predicates:

$$\mathrm{pia}_{generate}(\mathit{finding}(X), \mathit{causality}(\mathit{finding}(X), \mathit{hypothesis}(Y)), \mathit{hypothesis}(Y))$$

$$\mathrm{pia}_{select}(\mathit{hypothesis}(X), \mathit{preference}(Z), \mathit{diagnosis}(Y))$$

The descriptions of the inference actions *generate* and *select* are given in Figure 2.

It remains to define the different knowledge roles. *Causality* and *preference* provide domain knowledge and *finding* the case data for the inference layer. The inference layer is modelled as a *meta-language* of the domain layer in (ML)^2. This meta-relation enables the inference-

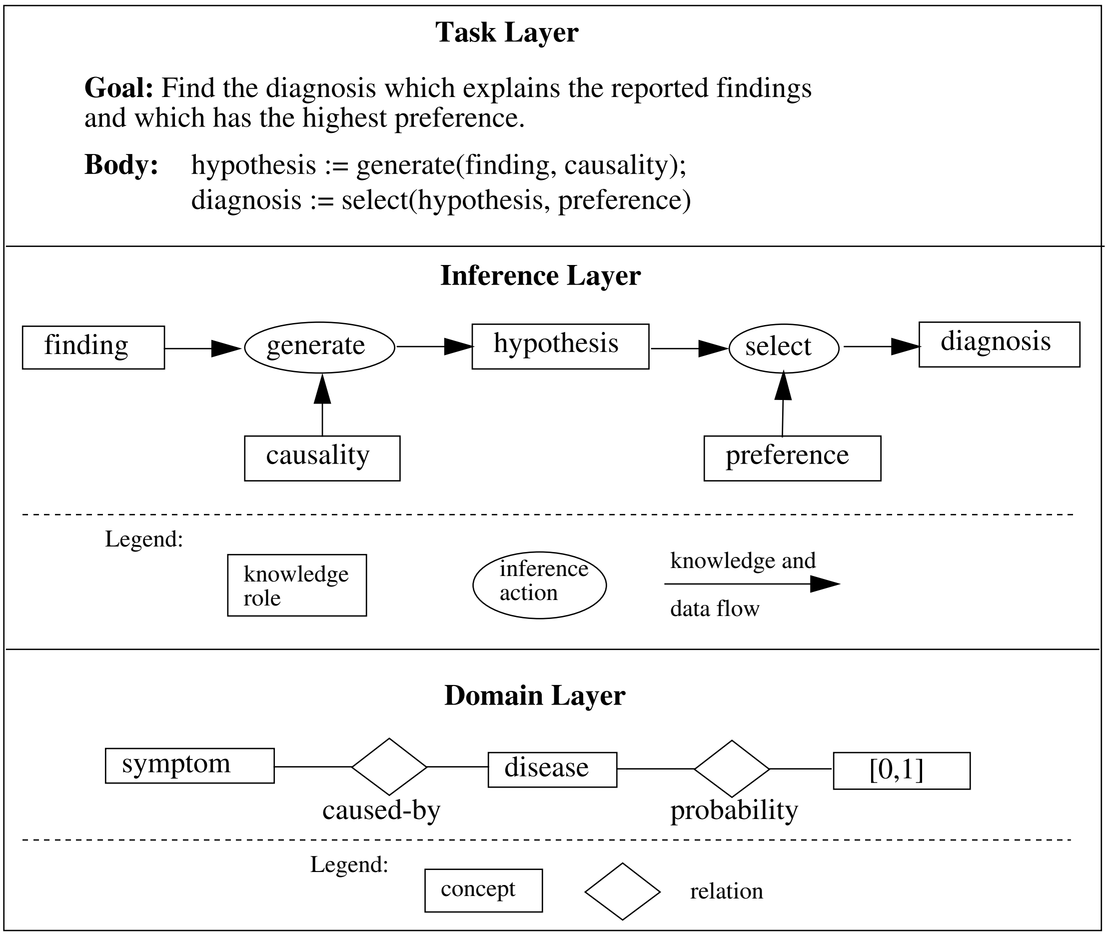

Fig. 1. A model of expertise for a simplified diagnostic task.

layer to specify properties of relations over domain-layer expressions (predicates and

theory *generate*

input roles *finding*, *causality*;

output roles *hypothesis*;

signature

sorts *finding-value, finding-name, causality-name, hypothesis-value, hypothesis-name;*

variables X : *finding-value,* Y : *hypothesis-value;*

functions

finding : *finding-value* $\longrightarrow$ *finding-name*,

causality : *finding-name* * *hypothesis-name* $\longrightarrow$ *causality-name*,

hypothesis : *hypothesis-value* $\longrightarrow$ *hypothesis-name*;

predicates $\mathrm{pia}_{generate}$: *finding-name* * *causality-name* * *hypothesis-name;*

axioms

$\mathrm{pia}_{generate}(\mathit{finding}(X), \mathit{causality}(\mathit{finding}(X), \mathit{hypothesis}(Y)), \mathit{hypothesis}(Y)) \leftarrow$

$\mathrm{input}_{finding}(\mathit{finding}(X)) \wedge \mathrm{input}_{causality}(\mathit{causality}(\mathit{finding}(X), \mathit{hypothesis}(Y)))$

endtheory

theory *select*

input roles *hypothesis*, *preference*;

output roles *diagnosis*;

signature

sorts

hypothesis-value, hypothesis-name,

diagnosis-value, diagnosis-name,

preference-pairs, preference-pair-set-value, preference-pair-set-name;

variables X : *hypothesis-value,* Y : *diagnosis-value,* Z : *preference-pairs;*

functions

hypothesis : *hypothesis-value* $\longrightarrow$ *hypothesis-name*,

preference : *preference-pair-set-value* $\longrightarrow$ *preference-pair-set-name*,

diagnosis : *diagnosis-value* $\longrightarrow$ *diagnosis-name*,

pref : *hypothesis-name* * *hypothesis-name* $\longrightarrow$ *preference-pairs*;

predicates

pia_{select} : *hypothesis-name* * *preference-pair-set-name* * *diagnosis-name;*

axioms

$\mathrm{pia}_{select}(\mathit{hypothesis}(X), \mathit{preference}(Z), \mathit{diagnosis}(X)) \leftarrow$

$\mathrm{input}_{hypothesis}(\mathit{hypothesis}(X)) \wedge$

$\mathrm{input}_{preference}(\mathit{preference}(Z)) \wedge$

$\neg(\exists Y : \mathrm{input}_{hypothesis}(\mathit{hypothesis}(Y)) \wedge \mathit{pref}(\mathit{hypothesis}(Y), \mathit{hypothesis}(X)) \in Z)$

endtheory

Fig. 2. Inference actions *generate* and *select* in (ML)^2.

functions). The expressions of object- and meta-language are connected by a naming relation and the truth values of formulae are connected by reflection rules. The input predicates $\text{input}_{finding}$, $\text{input}_{causality}$, and $\text{input}_{preference}$ used in the logical theories of the inference actions $\text{pia}_{generate}$ and pia_{select} are defined by reflection rules that connect truth in object- and meta-logic. As we abstract from this aspect of $(\text{ML})^2$ we will not go into any detail of this topic (cf. [van Harmelen & Balder, 1992]). The knowledge role *hypothesis,* however, 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 definition at all.

Quantified dynamic logic is used to specify dynamic control at the task layer. The *pia*-predicates, together with the test operator (of the form $pia_{name}\ ?$), are the elementary program statements. A history variable $V_{pia_{name}}$ is defined for each inference action that stores the input-output pairs for every execution step.

Four types of task-layer operations are available for each inference action pia_{name}: checking whether an instantiation exists ($has\text{-}solution\text{-}pia_{name}$), checking whether an instantiation has already been computed ($old\text{-}solution\text{-}pia_{name}$), checking whether more instantiations exist ($more\text{-}solution\text{-}pia_{name}$), and actually computing and storing a new instantiation ($give\text{-}solution\text{-}pia_{name}$):

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

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

$$give\text{-}solution\text{-}pia_{name}\ (I,O) =_{\text{def}} (more\text{-}solution\text{-}pia_{name}(I,O)?;\ V_{pia_{name}} := \langle (I,O) \mid V_{pia_{name}} \rangle$$

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. Note that $old\text{-}solution\text{-}pia_{name}$ and hence $V_{pia_{name}}$ is an administration for the non-deterministic execution of $give\text{-}solution\text{-}pia_{name}$ which is 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.

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

while $more\text{-}solution\text{-}pia_{generate}(finding(X),causality(finding(X),hypothesis(Y)),hypothesis(Y))$
do $give\text{-}solution\text{-}pia_{generate}(finding(X),causality(finding(X),hypothesis(Y)),hypothesis(Y))$
enddo
$give\text{-}solution\text{-}pia_{select}(hypothesis(X),preference(Z),diagnosis(X))$

Fig. 3. A task layer in $(\text{ML})^2$.

definition of the input predicate is the way in which $(ML)^2$ can be used to define data flow 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 3.

Dynamic Logic [Harel, 1984] uses *Kripke structures* to define a semantics for programs. A structure has the form $S = (D,F,P)$ consisting of a *domain* D, an interpretation F of the function symbols and an interpretation P of the predicate symbols. A *state over S* is a *function* s interpreting variables as elements of D. The interpretation of functions and predicates is fixed for all states. Let W denote the set of all states. Programs p are interpreted by binary relations between states. Formulas φ are interpreted by the collection of states for which they are true. For example,

$$I(p;q) = I(p) \circ I(q) = \{(s_0,s_2) \mid s_0,s_2 \in \mathrm{W} \wedge \exists\, s_1 \in \mathrm{W} \text{ with } (s_0,s_1) \in I(p) \wedge (s_1,s_2) \in I(q)\}$$

$$I(\varphi?) = \{(s,s) \mid s \in W \wedge \varphi \text{ is true in } s\}$$

$$I(Y := X) = \{(s_0,s_1) \mid s_0,s_1 \in \mathrm{W} \wedge s_1(Y) = s_0(X) \wedge \forall\, Z\,(Z \neq Y \rightarrow s_0(Z) = s_1(Z))\}$$

The most important program in $(ML)^2$ is *give-solution-pia*$_i$ which gives one possible solution. The essence of this elementary state transition in $(ML)^2$ is to apply the test operator *?* to the predicate pia_{name} which defines an inference action. $pia_{name}(I,O)?$ has as successor state a state that interprets (i.e., substitutes) the variables I,O in a way that fulfils $pia_{name}(I,O)$. In the successor state, this variable substitution is stored in the history variable of the inference action. Slightly simplified, we have the following pattern:

$$\begin{aligned} & I(P(X)?;Y := X) \\ & \quad = \{(s_0,s_1) \mid s_0 \in I(P(X)) \wedge (s_0,s_1) \in I(Y := X)\} \\ & \quad = \{(s_0,s_1) \mid P(X) \text{ is true in } s_0 \wedge s_1(Y) = s_0(X) \wedge \forall\, Z\,(Z \neq Y \rightarrow s_0(Z) = s_1(Z))\} \end{aligned}$$

2.3 KARL

The language KARL ([Fensel, 1995b], [Fensel, Angele & Studer]) 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., 1995], 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 sketch the domain layer and discuss the inference and task layers in KARL and sum up with a discussion of the semantics of dynamics.

L-KARL is used to describe the domain layer. It provides predicates, classes, class taxonomies, single- and set-valued attributes with domain and range restrictions, as well as multiple attribute inheritance for modelling terminological domain knowledge.

L-KARL is also used for specifying inference actions and knowledge 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). These roles are only accessible as input roles by inference actions. *Terminators* define a downward translation from the inference layer to the domain layer (giving write-access). These roles are only accessible as output roles by inference actions. *Stores* provide the input or output of inference actions. Therefore, they can be used as input and output roles by inference actions. Whereas views and terminators are used to link a domain layer with a generic inference layer, stores are used to model the data flow dependencies between inference actions. The definitions of the inference actions, stores, views, and terminators in our example are given in Figure 4.

P-KARL provides procedural control constructs for the task-layer. The primitive programs correspond to *calling an inference action*. 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 form subtasks like procedures in programming languages. The task layer of our example looks like this:

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. Views do not have a counterpart at the task layer because they do not have a dynamic interpretation (i.e., their interpretation is the same in each state). The value assignments of the variables that model stores and terminators are used to represent the current state of the reasoning process.

The logical language L-KARL has a minimal Herbrand model semantics [Lloyd, 1987]. Because we allow stratified negation in rules bodies we use a specific minimal model, i.e., the perfect Herbrand model, as the semantics of a set of clauses, cf.[Przymusinski, 1988]. Therefore, L-KARL does not use classical negation but a variant of the closed-world assumption that is common in approaches to logic programming and deductive databases.

P-KARL is a variant of dynamic logic using *Kripke structures* as semantics. A signature in dynamic logic consists of a set of function symbols and predicate symbols. An interpretation provides a domain or universe D, some functions $F = \{f^A{}_1, f^A{}_2,...\}$ over the domain used to interpret the function symbols, and some relations $P = \{P^A{}_1, P^A{}_2,...\}$ over the domain used to interpret the predicate symbols. The set of inference actions pia_{name} appears as a function symbol in the signature and each store $store_{name}$ appears as a predicate symbol $\varnothing(store_{name})$ in the signature.

The integration of the modal semantics of the task layer and the Herbrand models of L-KARL is as follows: the models of L-KARL are used to define an interpretation for a P-KARL language, i.e., the perfect Herbrand model of the set of clauses which define an inference action pia_{name} is used to interpret a function symbol pia_{name} occurring in assignments in P-KARL. Each store and each terminator is modelled by a (program) variable. The current state is represented by an assignment s of these variables. Notice, that a set of ground facts is assigned to each program variable. Slightly simplified, a transition is defined as:

$I(output\text{-}role(X) := pia_{name}(input\text{-}role)) =$
$\{(s_0,s_1) \mid s_1(output\text{-}role) = \text{perfect-Herbrand-model}(PIA_{name} \cup s_o(input\text{-}role))\}$
where PIA_{name} is the set of Horn clauses describing pia_{name}.

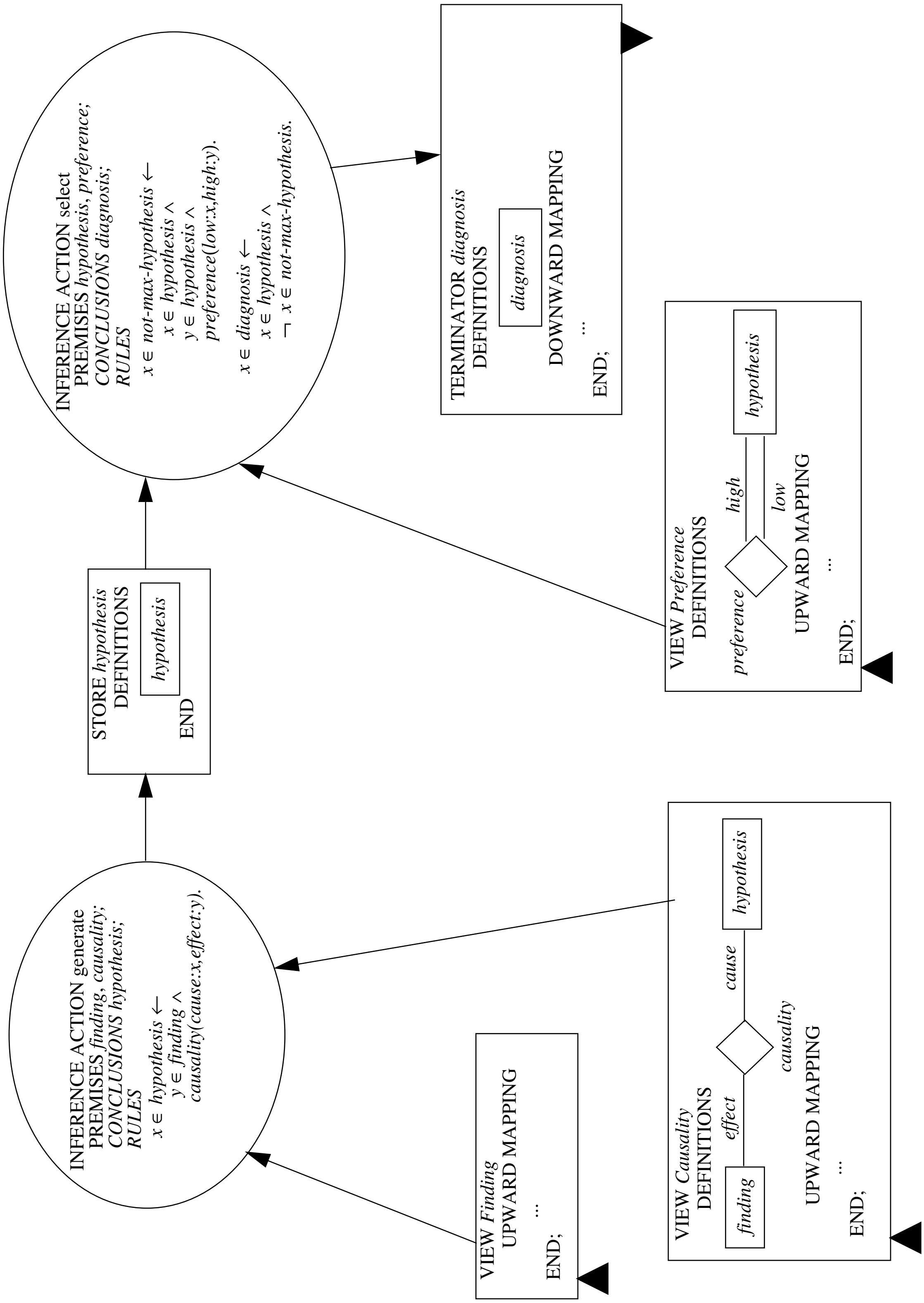

Fig. 4. An inference layer in KARL.

Finally $\emptyset(store_{name})$ is determined to be true for all states s with $s(store_{name}) = \emptyset$. The domain D is defined by the Herbrand base of the L-KARL language.

2.4 Design Rationales for MCL

In the following, we discuss our design rationales and their relations to the existing approaches. We discuss the following aspects for characterizing a reasoning process: (1) The *state* of a reasoning process, (2) the *history* of a reasoning process, (3) the *elementary* state *transitions*, (4) the *connection* of *states* and state *transitions*, and (5) *composed* state *transitions*.

2.4.1 The State of the Reasoning Process

Three choices arise in regard to the representation of a state of a reasoning process. First, whether its characterization is necessary at all, second whether its characterization is syntactic or semantic, and third whether its characterization should be local or global.

Is There a Notion of States. Abstract data types were developed in software engineering for the functional specification of software artifacts [Wirsing, 1990]. They should not make any commitments to the algorithmic process that realises the functionality. They define the functionality as a relation between input and output but have neither syntactically nor semantically the notion of a state. However, other approaches in software engineering, like VDM [Jones, 1990], Z [Spivey, 1992], and evolving algebras [Börger, 1995], use the notion of states for specifications. In Artificial Intelligence, problem solving is viewed as a search process through a state space. The problems tackled either do not have a complete functional specification or the functional specification defines a computationally hard problem that additionally requires the specification of a heuristic procedure that partly solves it. Therefore, approaches like ATMS [de Kleer, 1986] and situation calculus [McCarthy & Hayes, 1969] use states to specify the dynamics of a reasoning process.

Is This Notion Syntactic or Semantic. The situation calculus reifies the notion of state within first-order logic through the use of a special class of terms. Simplified, a predicate $p(x)$ is enriched by an argument that denotes states, i.e. $p(x,state)$, and the truth values of $p(x,state)$ can therefore be distinguished from the truth values of $p(x,succ(state))$. States are *syntactical* elements of the language in situation calculus. Conversely, dynamic logic provides a *semantical* notion of states. A state is characterized through a value assignment of all free variables. There is no syntactical notion that refers to a state. Therefore its semantics has to extend first-order models to a set of worlds that are used to interpret states.

Syntactical reification of states in the situation calculus is achieved by assigning names (i.e., terms that denote states) to them. This brings about the effect that two states that are identical except for having different names are regarded as different. In semantic-based approaches like dynamic logic, two states that have the same variable assignments to all variables cannot be distinguished, i.e., they are treated as equal. Syntactical versions like situation calculus require complex equality axioms to achieve the same.

Is the Characterization of States Local or Global. The *global* representation of states is quite natural for a monolithic sequential problem solver with a procedural control. Procedural control assumes one unique state at each moment of the entire reasoning process. Local representation of states is used for distributed problem-solving agents that cooperate during problem solving without a central control. Here, each component has an internal state. These internal states need not be uniquely related to internal local states of other components. Such approaches can be found in the areas of complex informations systems, distributed AI, and multi-agent systems (see [Jungclaus, 1993], [Weiß, 1995], [Brazier et al., 1995]).

Resume: $(ML)^2$ and KARL make the following choices according to our criteria: Both approaches are state-based, both approaches use the semantical notion of states in accordance with dynamic logic, and both approaches have one global state of the reasoning process. MCL makes precisely the same design decisions. However, it differs in the way a state is represented. As mentioned above, a state is represented in dynamic logic by value assignments of all open variables. MCL uses a richer structure to represent a state in accordance to the *states-as-algebras* setting of MLCM. In this setting, an algebra (i.e., a rich data structure) instead of a flat list of variables is used to present a state. A state is characterized by an interpretation of all predicates and functions. One advantage of this is that it allows us to overcome a non-intuitive aspect of dynamic logic where the same variables are used in a logical and in a dynamic (i.e., state-based) sense (see Section 2.4.4).

In Section 5 we will also discuss approaches in knowledge engineering that made different design decisions: TFL [Pierret-Golbreich & Talon, 1996], which does not use the notion of states at all, and DESIRE [van Langevelde et al.,1993], which uses a local representation of states to express composed and distributed problem solving.

2.4.2 The History of the Reasoning Process

Two states are equal in a state-based approach if they do not differ in any property. In a history-based approach they are different if they were achieved through different *paths* of the reasoning process, i.e. the history of the reasoning process is part of the state description. The history of the reasoning process is necessary for software that models real-world processes, like in robotics and work-flow management systems, or in strategical reasoning about different choices. When modelling the movement of a robot it is not only necessary to end up in a proper terminal state. There are also important constraints on intermediate states and their proper sequence. The situation is similar in work-flow management systems, where decentralised processes need to be synchronised properly. Finally, in strategic reasoning we reason about different paths of the reasoning process. For example, when a system runs into a dead end in its reasoning process it needs the information about the path that lead to this dead end in order to choose more appropriate reasoning possibilities. Examples for history-based approaches are situation calculus, which reifies history information syntactically using term structures, Transaction logic [Bonner & Kifer, 1993], which provides a path semantics to express history, and DESIRE [Treur, 1994], which uses temporal models as the semantics of reasoning paths.

The *KADS model of expertise* represents the control of the reasoning process of a KBS at the task layer. A simple procedural control language is provided for this purpose (cf. [Schreiber et al., 1994]). The restriction to simple control also implies that we will not aim at specifying the history of the reasoning process with MCL. Like KARL, MCL has no notion of history. This restricts our abilities for an elegant representation of *strategic* reasoning including the reasoning about earlier states of the problem-solving process, but this goal is beyond the scope of our approach. This type of knowledge was allocated at a different layer (the *strategic layer*) in earlier versions of the KADS model of expertise.

The decisions in $(ML)^2$ to characterize a state by the set of the *local* histories of all inference actions and the *syntactic* integration of *histories* via lists that implement state traces in the *state*-oriented semantical framework of dynamic logic are rather non-standard and are not integrated into MCL.[4]

2.4.3 Elementary State Transitions

Inference actions are modelled as relations in $(ML)^2$ (see Section 2.2) and as functions in KARL (see Section 2.3). In $(ML)^2$, each inference action defines a relation that is used to interpret a predicate symbol used in a test operation in dynamic logic. In KARL, each inference action defines a function which is used to interpret a function symbol used in an assignment in dynamic logic. $(ML)^2$ changes a state by selecting precisely one new instantiation of a predicate (i.e. for the given state the predicate is true for one ground variable assignment and false for all others). KARL changes a state by determining all instantiations of a predicate to be true that follow from the logical theory of an inference actions and its input. Both types of inferences appear in formalized KADS models. We see this in our running example. The inference action *generate* should generate all possible hypotheses and the inference action *select* should select one of them as a diagnosis. It is possible to express one inference type by the other type but it results in an artificial modelling. A loop of updates at the task layer in $(ML)^2$ is required to simulate an update in KARL. KARL needs a random-selection predicate in the definition of an inference action to simulate an update in $(ML)^2$ that non-deterministically selects one instantiation of a predicate. As a consequence, MCL provides both types of state transitions.

We call both update types *bulk* updates as they change the complete extension of a predicate in one step. Each ground literal of a predicate symbol is reevaluated by such a state transition. MLCM provided only *point-wise* modification of constants, functions, and predicates. The extension to MCL is precisely concerned with introducing elementary state transitions of a higher grainsize that can directly express such bulk-updates.

2.4.4 Connecting State Transitions with States

$(ML)^2$ uses the history variables in the definition of the input predicates and the test operator *?* of dynamic logic that transform a formula into a state transition to combine the value assignments of logical and program variables. V_{pia} collects the results of an inference action and can provide them to another inference action via the definition of an corresponding input predicate. This solution relies on the identification of logical and dynamic (state-dependent) variables in dynamic logic. Take as an example the following formula from dynamic logic; the evaluation of the existential quantification more or less “undoes“ the states modification of the program within the modal operator.

$$[x := 3]\ (\exists\ x\ (x = 2))$$

A further problem is that $(ML)^2$ requires modelling constructs that are not mentioned in the conceptual modelling context of the KADS model of expertise like *has-solution-pia*, *old-solution-pia*, *more-solution-pia*$_n$, *give-solution-pia*, as well as history variables V_{pia} and complex input predicate definitions for each inference action to connect the specification static specifications with dynamic state change.

4. The history information for inference actions stored in the history variables V_{pia_i} is used in $(ML)^2$ to prevent inference actions from always deriving the same output and is not used for strategic reasoning (compare Sections 2.2 and 4).

KARL uses a somewhat non-standard approach to achieve the combination of the functional specification of state transitions and states. The minimal model semantics of the set of clauses that define a state transitions is used as an interpretation of the corresponding function symbol in the dynamic logic.

An important motivation of our exercise is to find a better solution for this integration. We want to separate logical variables used in the definition of elementary transitions and program variables which express the dynamic state of the reasoning process without externalizing the definition of a state transition as an interpretation of function symbols. In part we will follow the intuition of KARL, where logical variables used to characterize inference actions and program variables used to memorize a state are distinguished. Part of a state characterization are the literals that hold. However, this is not achieved by assigning a set of true literals to a program variable but by directly using an algebra to express a state. State changes are expressed by changes of this algebra. We will use the *states as algebras* approach for this purpose. As a consequence, we do not even need program variables in our framework.

2.4.5 Composed State Transitions

The task layer of a KADS model of expertise defines *sequence*, *alternative*, and *loops* of inference actions. A specification language has to provide these means to form composed transitions.

2.4.6 Resume

A language for specifying the reasoning process of KBSs, based on the KADS model of expertise must provide the following:

- It must be possible to express the global state of the reasoning process.
- It is not required to represent the history of the reasoning process.
- It must be possible to only characterize complex sub-steps functionally without making commitments to their algorithmic realization.
- The description of state transitions must be easily and intuitively related to state changes.
- Finally, it must be possible to express algorithmic control over the execution of sub-steps.

In the following we present our solution for these goals.

3 Modal Change Logic (MCL)

MCL is a new version of MLCM (Modal logic of Creation and Modification, see [Groenboom & Renardel de Lavalette, 1994], [Groenboom, 1997]). MLCM was developed for reasoning about dynamic aspects of the specification language COLD (see [Feijs & Jonkers, 1992]); MCL is a generalization where local (i.e. *point-wise*) modification of functions and predicates is generalized to global modification (also called bulk update) and a choice quantifier is added. This generalization was necessary to express complex state transitions that are defined by an inference actions in the KADS model of expertise.

First, we put MCL in the perspective of multimodal dynamic logics. Modal logic is an extension of (propositional or predicate) logic with the unary sentential operator $\Box$, where $\Box A$

traditionally has the informal meaning *it is necessary that A*. An early reference is [Lewis & Langford, 1932]. In [Kripke, 1959], Kripke developed the *possible-worlds* semantics, according to which the formula $\Box A$ is true in some world w iff A holds in all worlds that are accessible from w via the relation R. There are many modal logics, each corresponding with a particular class of accessibility relations: e.g., the logic axiomatized by $\Box A \to A$ corresponds with reflexive relations (satisfying $\forall x\ (xRx)$, and $\Box A \to \Box\Box A$ with transitive relations, where $\forall x,y,z\ (xRy \wedge yRz \to xRz)$.

Multimodal logics are logics with more than one modal operator. Examples are temporal logics (with two modalities, one for the future, one for the past), multi-agent epistemic logic (one modality for the knowledge of each of the agents), and dynamic logic. In the latter, modalities are associated with programs, with the intended meaning *after the execution of the program*. The first formulation is by Pratt (in [Pratt, 1976]) using an idea of his student R.C. Moore. They investigated Floyd-Hoare logic, which features the expression $A\{\alpha\}B$ with the intended meaning *if A, then after doing* α *B holds*. In dynamic logic, this becomes $A \to [\alpha]B$. Surveys of dynamic logic are [Goldblatt, 1992] and [Harel, 1984].

Dynamic logic is often presented with the variable assignments $x{:=}t$ as its atomic programs. In MCL however, the atomic programs are $f{:=}\lambda x.t$, $p{:=}\lambda x.A$ (changing the interpretation of a function and a predicate, respectively) and NEW (which creates a new object and makes new refer to it). The first two types of programs generalize the program statements $f(s) := t$, $p(s) :\leftrightarrow A$ for *point-wise* function and predicate modification, which were introduced in MLCM (see [Groenboom & Renardel de Lavalette, 1994]), inspired by the definition of the specification language COLD (see [Feijs et al., 1987], [Feijs & Jonkers, 1992]). Point-wise function modification was already dealt with in [Pratt, 1976] (where it was called array assignment) and it is also a vital ingredient of Evolving Algebras (see [Gurevich, 1994]).

For the composition of programs, most dynamic logics (including MCL) contain the constructs of sequential composition (;), non-deterministic choice ($\cup$), iteration (*) and test (A?, where A is some formula). Some usual program statements can be defined using these constructs:

if A **then** α **else** β	=	$(A?;\alpha) \cup (\neg A?;\beta)$
while A **do** α	=	$(A?;\alpha)^*;\neg A?$

Moreover, MCL contains the choice quantifier $\cup x$. When applied to some program α, the meaning of the resulting program is: do α for some non-deterministically chosen value of x.

In the following we introduce the syntax, semantics, and axiomatization of MCL.

3.1 Syntax of MCL

Signatures Σ are collections of (function or predicate) symbols σ, with arity $\#\sigma \in N$. Predicate symbols are denoted by $p,q,r,\ldots$, function symbols by $f,g,h,\ldots$, and nullary function symbols (usually called constants) by $a,b,c,\ldots$. VAR is a countably infinite set of variable symbols, denoted by $x,y,z,\ldots$. The syntax of MCL, consisting of the syntactical categories TERM (terms), PROG (programs) and FORM (formulae), is defined by:

TERM $\quad t ::= \quad x \mid \uparrow \mid \mathsf{new} \mid f(t_1,\ldots,t_{\#f})$

PROG $\alpha ::=$ NEW $| f := \lambda x_1,..., x_{\#f}.t \mid p := \lambda x_1,..., x_{\#p}.A \mid$
$A? \mid \alpha;\alpha \mid \alpha \cup \alpha \mid \alpha^* \mid \cup x.\alpha$

FORM $A ::= (t = t) \mid p(t_1,..., t_{\#p}) \mid A \wedge A \mid \neg A \mid \forall x A \mid [\alpha]A$

For the sake of simplicity, we assume here and later that functions f and predicates p are unary.

$\top$, $\bot$, $\vee$, $\to$, $\leftrightarrow$, $\exists x$, $\langle\alpha\rangle$, are defined as usual. We also define weak equality and a definedness predicate:

$$t\downarrow \quad = \quad (t = t)$$

$$s \simeq t \quad = \quad ((s\downarrow \vee t\downarrow) \to s = t)$$

Substitution of a term for all free occurrences of a variable in a term, formula or program (denoted $(t/x)A$ etc.; so, e.g., $((f(y)/x)p(g(x,y)) = p(g(f(y),y)))$ is defined as usual (renaming bound variables in order to prevent variable clashes). However, we have to be careful because it may possible to substitute a term at an occurrence in the scope of a program that changes one or more signature elements of that term (e.g. $(c/x)([c{:=}f(c)](g(x){=}x))$. A substitution where this does *not* occur is called *safe*.

3.2 Semantics of MCL

MCL is interpreted in models $M = \langle U,V, \bigstar,W \rangle$, where U is the *basic universe*, $V = \{v_n \mid n \in N\}$ is the *store* of objects that can be created, $\bigstar$ is the undefined object, and W is a collection of *worlds*. We assume that $(U\cup\{\bigstar\})\cap V=\{v_0\}$ and define:

$V_n =_{\text{def}} \{v_m \mid m \leq n\}$, as an initial segment of the state.

$U_M =_{\text{def}} U \cup V \cup \{\bigstar\}$, as the full universe of M.

Every $w \in W$ is of the form $w = \langle n_w,I_w \rangle$, with n_w being the number of store elements and I_w an interpretation of the signature elements in the *local universe*

$$U_w = U_{n_w} = \mathrm{U} \cup V_{n_w} \cup \{\bigstar\}.$$

We shall write σ_w for $I_w(\sigma)$. Thus every world is a model of the first-order fragment of MCL. Extensions and updates of a world $w = \langle n,I\rangle$ are defined as follows ($F \in (U_w \to U_w)$, $P \subseteq U_w$):

- w^+ is the extended world $\langle n{+}1,I^+\rangle$ where I^+ satisfies $I^+(\sigma)(v_{n+1}) = I(\sigma)(\bigstar)$, $I^+(\sigma)(u) = I(\sigma)(u)$ for all $u \in U_w$;
- $w[f \mapsto F]$ is the updated world $\langle n,I'\rangle$, satisfying $I'(f) = F$, $I'(\sigma) = \mathrm{I}(\sigma)$ if $\sigma \neq f$;
- $w[p \mapsto P]$ is the updated world $\langle n,I'\rangle$, satisfying $I'(p) = P$, $I'(\sigma) = I(\sigma)$ if $\sigma \neq p$.

We postulate that W **is closed under extensions and updates**. Observe that any W can always be extended to meet this requirement.

ASS = VAR $\to U_M$ is the collection of assignments. Point-wise modification $a[x \mapsto u]$ (where $x \in$ VAR, $u \in U_M$) of $a \in$ ASS is defined as usual. An assignment a can be restricted in a world w to a_w : VAR $\to U_M$ as follows:

$$a_w(x) = a(x) \quad \text{if } a(x) \in U_w$$
$$= \bigstar \quad \text{if } a(x) \notin U_w$$

$[\![t]\!]_{w,a}$, the interpretation of term t in world w with assignment a, is defined by:

$$[\![x]\!]_{w,a} \quad = \quad a_w(x)$$

$$[\![\uparrow]\!]_{w,a} \quad = \quad \star$$

$$[\![\mathsf{new}]\!]_{w,a} \quad = \quad v_{n_w}$$

$$[\![ft]\!]_{w,a} \quad = \quad f_w([\![t]\!]_{w,a})$$

$w,a \models A$ (the interpretation of formula A in world w with assignment a) and $R_{\alpha,a}$ (the accessibility relation of program α w.r.t. assignment a) are defined simultaneously:

$$w,a \models (s = t) \quad =_{\text{def}} \quad [\![s]\!]_{w,a} = [\![t]\!]_{w,a} \neq \star$$

$$w,a \models pt \quad =_{\text{def}} \quad p_w([\![t]\!]_{w,a}) = \textbf{true}$$

$$w,a \models \neg A \quad =_{\text{def}} \quad \textbf{not}\ (w,a \models A)$$

$$w,a \models A \wedge B \quad =_{\text{def}} \quad w,a \models A \ \textbf{and}\ w,a \models B$$

$$w,a \models \forall x A \quad =_{\text{def}} \quad \textbf{forall}\ u \in (U_w \setminus \{\star\})\ (w,a[x \mapsto u] \models A)$$

$$w,a \models [\alpha]A \quad =_{\text{def}} \quad \textbf{forall}\ w' \in W\ (wR_{\alpha,a}w' \Rightarrow w',a \models A)$$

One new element is added to the universe:

$$R_{\mathsf{NEW},a} \quad =_{\text{def}} \quad \{(w,w^+) \mid w \in W\}$$

Modification of the values of a function:

$$R_{f:=\lambda x.t,a} \quad =_{\text{def}} \quad \{(w,w[f \mapsto \lambda u \in U_w.[\![t]\!]_{w,a[x \mapsto u]}]) \mid w \in W\}$$

Modification of the truth values of a predicate:

$$R_{p:=\lambda x.A,a} =_{\text{def}} \quad \{(w,w[p \mapsto \{u \in U_w \mid w,a[x \mapsto u] \models \mathrm{A}\ \}]) \mid w \in W\}$$

A program is executed for one value

$$R_{\cup x.\alpha,a} \quad =_{\text{def}} \quad \{(w,w') \mid \textbf{exists}\ u \in U_w,\ wR_{\alpha,a[x \mapsto u]}w'\}$$

The following four transition relations are standard relations of dynamic logic

$$R_{A?,a} \quad =_{\text{def}} \quad \{(w,w) \mid w,a \models A\}$$

$$R_{\alpha;\beta,a} \quad =_{\text{def}} \quad R_{\alpha,a} \circ R_{\beta,a}$$

$$R_{\alpha\cup\beta,a} \quad =_{\text{def}} \quad R_{\alpha,a} \cup R_{\beta,a}$$

$$R_{\alpha^*,a} \quad =_{\text{def}} \quad \mathrm{R}^*_{\alpha,a} \text{ (i.e., the transitive closure of } \mathrm{R}_{\alpha,a})$$

3.3 Axiomatization of MCL

The axioms are:

Taut All tautologies of propositional logic

Eq $x \simeq y \rightarrow y \simeq x$

$x \simeq y \rightarrow fx \simeq fy \wedge (px \leftrightarrow py) \wedge (x \simeq z \leftrightarrow y \simeq z)$

Undef $\neg(\uparrow\downarrow)$

Inst	$(\forall xA \wedge x\!\downarrow) \to A$	
Atom	$[\pi]\neg A \leftrightarrow \neg[\pi]A$	(π atomic)
C1	$x = y \leftrightarrow [\mathsf{NEW}](x = y \neq \mathsf{new})$	
C2	$px \leftrightarrow [\mathsf{NEW}]\, px$	
C3	$[\mathsf{NEW}]\ (\mathsf{new}\!\downarrow \wedge fx \neq \mathsf{new} \wedge f(\mathsf{new}) \simeq f(\uparrow) \wedge (p(\mathsf{new}) \leftrightarrow p(\uparrow)))$	
FM1	$A \leftrightarrow [f{:=}\lambda x.t]\, A$	for all $f \notin \mathrm{sig}(A)$
FM2	$[f{:=}\lambda x.t]\, fx = y \leftrightarrow t = \mathrm{y}$	(y not free in t)
FM3	$\forall y\ [f{:=}\lambda x.t]A \leftrightarrow [f{:=}\lambda x.t](\forall y\, A)$	($x \equiv y$ or y not free in t)
PM1	$B \leftrightarrow [p{:=}\lambda x.A]B$	for all $p \notin \mathrm{sig}(B)$
PM2	$[p{:=}\lambda x.A]px \leftrightarrow A$	
PM3	$\forall y\ [p{:=}\lambda x.A]B \leftrightarrow [p{:=}\lambda x.A](\forall y\, B)$	($x \equiv y$ or y not free in A)
?AX	$[A?]B \leftrightarrow (A \to B)$	
;AX	$[\alpha;\beta]A \leftrightarrow [\alpha][\beta]A$	
∪AX1	$[\alpha\cup\beta]A \leftrightarrow ([\alpha]A \wedge [\beta]A)$	
***AX**	$[\alpha^*]A \leftrightarrow (A \wedge [\alpha][\alpha^*]A)$	
∪AX2	$[\cup x.\alpha]A \leftrightarrow \forall x[\alpha]A$	(x not free in A)

$\Gamma \vdash A$ (A is derivable from Γ; Γ may be omitted when empty) is defined inductively by

AX	$\vdash A$ if A is a safe substitution instance of an axiom
MP	$A, A \to B \vdash B$
INF	$\{[\alpha^n]A \mid n \in N\} \vdash [\alpha^*]A$
W	if $\Gamma \vdash A$ then $\Gamma,\Delta \vdash A$
CUT	if $\Gamma \vdash A$ for all $A \in \Delta$ and $\Gamma,\Delta \vdash B$ then $\Gamma \vdash B$
DED	if $\Gamma, A \vdash B$ then $\Gamma \vdash A \to B$
UG	if $\Gamma, x\!\downarrow\ \vdash A$ and x does not occur free in Γ, then $\Gamma \vdash \forall xA$
NEC	if $\Gamma \vdash A$ then $[\alpha]\Gamma \vdash [\alpha]A$

Soundness is proved straightforwardly, although some lemmata involving substitution and the frame property are needed. We claim that completeness also holds: a proof (a nontrivial variant of the Henkin construction, see [Henkin, 1949]) will appear elsewhere.

4 Using MCL to Formalize Other Approaches

In the following, we discuss first how MCL can be used to formalize the reasoning behavior of KBSs in a KADS-oriented style. We do this by showing for a number of KADS-oriented languages how their state transitions can be expressed with the operators of MCL. Second, we illustrate the generality and power of our approach by relating it with other areas of research. We discuss how MCL can be used to formalize evolving algebras and database update languages.

4.1 Formalizing KADS languages

In this subsection we will discuss the formalization of MLPM, $(ML)^2$, and KARL with MCL.

4.1.1 MLPM

The Modal Logic of Predicate Modification (MLPM) was introduced by [Fensel & Groenboom, 1996] for formalizing KADS languages like KARL and $(ML)^2$. MLCM was taken as a departure point for this exercise. For reasons of simplicity, only the predicate modification operator of MLCM was taken. This predicate operator had to be generalized to two types of bulk-updates because KADS inference action may modify a complete extension of a singleton predicate whereas MLCM only offered a point-wise update. The two new operators were:

- $p := \lambda.xA$, that corresponds to the λ-operator of MCL restricted to unary predicates,
- $p := \varepsilon.xA$, defining (non-deterministically) p true for exactly one x satisfying A.

The ε operator of MLPM can be expressed by the choice quantifier of MCL:

$$p := \varepsilon.xA \qquad \equiv \qquad \cup x.(A?;\, p := \lambda y.(x = y))$$

(y fresh); as a consequence, it has the following semantics:

$$R_{p:=\, x.\varepsilon A,a} \quad = \quad \{(w, w[p \mapsto \{u\}]) \mid w \in W,\, u \in U_w,\, w,a[x \mapsto u] \models A\}$$

and the axioms are:

SP1	$\langle p := \varepsilon x.A \rangle\, px \leftrightarrow A$	
SP2	$[p := \varepsilon x.A]\, \exists! x\, px$	
SP3	$(\exists x\, A \rightarrow B) \leftrightarrow [p := \varepsilon x.A]\, B$	$p \notin sig(B)$
SP4	$\forall y\, [p := \varepsilon x.A] B \leftrightarrow [p := \varepsilon x.A](\forall y\, B)$	y not free in A

The generalization of MLCM and MLPM which we achieved with MCL provides the advantage that it reintegrates the generalizations of MLPM into the general setting of MLCM. Therefore the introduction of new elements and (global) function updates can also be expressed. In consequence, we get a unified approach that covers many existing approaches from knowledge engineering and other research areas as shown below.

4.1.2 $(ML)^2$

The choice quantifier ($\cup\, x.\alpha$) with

$$\alpha \quad \equiv \quad ((x/y)pia_{name}(x,...)?;\, output\text{-}role_{name} := \lambda y.(x = z))$$

of MCL captures the core of the state transitions in $(ML)^2$. The singleton predicate modelling the output role is true for one ground instantiation (modulo equality) and false for all others. This can be used to model non-deterministic selection. We have decided not to hardwire the mechanism *give-solution-pia* of $(ML)^2$ directly in MCL because there are some problems related to this construct as it can behave in a non-intuitive manner. For example, a deterministic inference action like *multiplication* fails if it gets the same input values a second time.

4.1.3 KARL

The λ-operator of MCL applied to a predicate can be used to model the bulk-update of KARL. The call of an inference action in KARL, for example, the inference action *generate*

$hypothesis := generate(finding)$
with
$\forall x,y(finding(x) \wedge causality(x,y) \rightarrow hypothesis(y))$
as logical theory defining the inference.

can be expressed in MCL as

$hypothesis :\leftrightarrow \lambda y.(\exists x(finding(x) \wedge causality(x,y)))$.

In the above statement the value of the output role is erased. If we would like to *extend* the extension of a role, we can formulate this as:

$role :\leftrightarrow \lambda y.(p(y) \vee role(y))$

KARL requires us to artificially introduce two inference actions (one select and one copy step) and an additional placeholder when facts of an input role should be deleted. The KARL statement

$placeholder := select(input); input := copy(placeholder)$

is formulated in MCL as:

$input :\leftrightarrow \lambda x.(input(x) \wedge select(x))$

However, a significant difference remains between MCL at the one hand and KARL at the other hand. KARL uses minimal and perfect Herbrand model semantics [Lloyd, 1987], [Przymusinski, 1988] to evaluate a set of clauses. Based on this reasoning with the closed-world assumption, negative literals and (in the case of stratification) positive literals of higher strata can be derived from a set of clauses in KARL which are not a logical consequence in the standard model-theoretical framework of first-order logic upon which MCL is based.

Minimal-model semantics of a logical program can be expressed in MCL along the lines of the operational semantics of fixpoint computation. When a fixpoint is reached after a finite number of iterations (which is assumed in KARL, see [Fensel, Angele & Studer]), this can be expressed in MCL as a finite number of applications of a program operator. For the unary case this goes as follows: Let H be a logical program, the formula A the disjunction of all clauses in H, p a unary predicate symbol of H, and $\lambda p.\lambda x.A$ a new predicate operator associated with H. Then p equals the fixpoint after termination of the following program α^H with:

$(p := \lambda x.\bot); (p := \lambda x.A)^*$

i.e. we have

$fix(H)x \leftrightarrow \langle\alpha^H\rangle px$

In order to deal with non-unary predicate operators, MCL has to be extended by a kind of *parallel* construct. We illustrate this with a binary predicate operator given by $\lambda p.\lambda x.A$, $\lambda q.\lambda y.B$. The required program α^H then becomes

$$(p := \lambda x.\bot,\ q := \lambda y.\bot);\ (p := \lambda x.A,\ q := \lambda y.B)^*$$

The parallel construct is denoted by a comma (,). The idea is that it can be applied on two programs which do not modify the same signature element(s) (cf. [Groenboom, 1997]).

4.2 Using MCL to Formalize Evolving Algebras

The two basic concepts of *evolving algebras* [Gurevich, 1994] are *states* and *state transitions*. As in COLD, a *state* is modelled by one static algebra. Let Υ be a signature, i.e. a finite collection of function names with given arity (the 0-ary function names model constants). A static algebra of a signature Υ is a nonempty set S together with interpretations on S of functions names in Υ. Such a static algebra defines one possible world (i.e., possible state). *Transitions* between states can be expressed by *function updates* of the form $f(t_1,\ldots,t_r) := t$. These updates can be qualified by guards which express preconditions for their application. The point-wise modification corresponds to the grainsize of the transition in MLCM. Evolving algebras also provide means to specify parallel algorithms that can be used to express our type of bulk-updates (i.e. the *choose* and the *range* constructs, cf. [Gurevich, 1994]). The formalization of evolving algebras in MCL (see table 1) is inspired on the work presented in Figure [Groenboom & Renardel de Lavalette, 1995].

Table 1. The elementary state transitions of evolving algebras

Evolving Algebras	characterization	MCL
$f(s){:=}t$	sets f at s to t	$f{:=}\lambda x.$(if $x{=}s$ then t else fx fi)
if A then r	execution of r guarded by A	$A?;r$
choose x in V do r	execute r for a value x from V	$\cup x.r$
import x in V do r	modify universe V with new object x and perform r	NEW;$\cup x(x{=}new)?;r$
range x in V do $f(x){:=}t$	let x range over the objects in V and perform update of f	$f{:=}\lambda x.t$

An evolving algebra program is a set of rules α_ι. The execution of such a program (called a run in [Gurevich, 1994]) is defined as the infinite execution of nondeterministically chosen rules α_ι. We can encode this in MCL as:

$$\text{EA} \quad = \quad (\cup x.\alpha_x)^*$$

Although the presented formalization captures the core of evolving algebras, some expressions in evolving algebras are not covered. MCL is an untyped logic, therefore we cannot model the different universes of evolving algebra directly. This requires however only a syntactic extension of MCL in order to mimic evolving algebras more directly. More involved is the modelling of the concurrent assignment of evolving algebras. In [Groenboom & Renardel de Lavalette, 1995] and [Groenboom, 1997] defines the parallel execution operator of evolving algebras “,“ is defined in terms of the simultaneous composition operator $\oplus$. Currently, this operation in not supported by MCL.

Two main differences exist between evolving algebras and our approach. First, evolving algebras do not aim at a formal specification with formal syntax, semantics, and automated proof support. Instead, they provide a semiformal mathematical notation for definitions and proofs in a textbook-like style (cf. [Börger, 1995]). Second, our approach provides procedural vocabulary to express control over the execution of transitions. Evolving algebras do not provide a vocabulary to specify such composed state transitions. The way control is specified in evolving algebras is close to the spirit of production rule systems. To put it in simple words, evolving algebras provide a set of local transitions rules and a rule interpreter built into the semantics of evolving algebras that selects the next firing rules to be applied.

4.3 Using MCL to Formalize Database Update Languages

The declarative specification of the static aspects of databases is well-established field. Ongoing work deals with the problem of defining clean and declarative characterizations of updates of databases. PDDL [Spruit et al., 1995] and DDL [Spruit et al., 1993] use different variants of dynamic logic for providing a logical characterization of the dynamics of databases. In the following, we will show how MCL can be used to express the state transitions introduced by these languages.

4.3.1 PDDL

Propositional dynamic database logic (PDDL) defines a variant of propositional dynamic logic by restricting elementary state transitions (i.e., elementary programs) to two pre-defined types. The (point-wise) update of the truth value of one proposition and the bulk update of the truth values of a set of propositions according to the minimal Herbrand model of a set of propositional Horn clauses. A state is described by the truth values of all propositions, and complex transitions can be built using the normal means of dynamic logic. PDDL is close in spirit to KARL when we abstract from all conceptual details and the fact that PDDL uses propositional dynamic logic only. Like KARL, PDDL uses a minimal Herbrand model of a set of clauses to define elementary updates and the operational fragment of KARL (see [Angele, 1993], [Fensel, Angele & Studer]) is restricted to finite Herbrand models (aside from some built-in types) and therefore has similar expressive power. Table 2 provides the transition types of PDDL and their counterpart in MCL.

Table 2. The elementary state transitions of PDDL

PDDL	characterization	MCL
I_p	sets the proposition p to true	$p{:=}\top$
D_p	sets the proposition p to false	$p{:=}\bot$,
$I^H{}_p$	sets p to true and computes the minimal model of $H \cup \{p\}$	$p{:=}\top;\ \alpha^H$
$D^H{}_p$	sets p to false and computes the minimal model of $H \cup \{\neg p\}$	$p{:=}\bot;\ \alpha^H$

4.3.2 DDL

Dynamic database Logic (DDL) [Spruit et al., 1993] extends the ideas of PDDL to the first-order case. Again two main types of predefined updates (i.e., elementary programs) are provided:

- Updating the truth values of all ground literals over a predicate symbol according to the truth values of the corresponding variable assignments of a first-order formula and
- the non-deterministic selection of one of the variable assignments that makes a formula true.

Aside from some details, both types of updates are similar to the λ- and ε-operator of MLPM. Complex transitions can be constructed using the normal means of dynamic logic. As in MLPM, a state is described by an interpretation of the predicate symbols. [Spruit et al., 1993] define DDL without function symbols and provide a complete axiomatization under the domain closure and unique naming assumptions (thus the expressive power of the language is restricted to the propositional case). Table 3 provides the transition types of PDDL and their counterpart in MCL.

Table 3. The elementary state transitions of DDL

DDL	characterization	MCL
$\&xIpt$ where A	inserts pt for all terms t that satisfy A	$p{:=}\lambda y.(py \vee \exists x(A \wedge y = t))$
$\&xDpt$ where A	deletes pt for all terms t that satisfy A	$p{:=}\lambda y.(py \wedge \neg\exists x(A \wedge y = t))$
$\&xUpt{\to}t'$ where A	replace pt by pt' for all t' for which A is true	$p{:=}\lambda y.($ $(py \wedge \neg\exists x(A \wedge y = t)) \vee$ $\exists x(pt \wedge A \wedge y = t'))$
$ft := t'$	changes f for the argument t to t'	$f{:=}\lambda x.$if $x{=}t'$ then t else fx fi
$+x\ \alpha$ where A	execute α for one assignment for x where A is true	$\cup x.(A?;\alpha)$

5 Related Work

The specification language TFL [Pierret-Golbreich & Talon, 1996] applies abstract data types to specify the functionality and the reasoning process of a KBS. Abstract data types are applied to specify domain and inference knowledge using loose semantics. Procedural control is specified by so-called process modules which incorporate the control expressions as operations into the framework of abstract data types. Test, sequence, choice, and iteration are specified as operations and axioms are used to further specify these operators (see Figure 5) as in *process algebra* [Baeten & Weijland, 1990]. The main difference to our approach is that TFL neither provides a syntactical nor a semantical notion of the *state* of the reasoning process and does not provide a predefined set of elementary state transitions. Therefore, TFL has the frame problem as the situation calculus [McCarthy & Hayes, 1969]. For each elementary action the specifier must specify what it changes and what it keeps unchanged. In state-based approaches like dynamic logic and MCL this is already provided by the semantics and axiomatization of elementary state transitions. That is, the semantics of an elementary transition like $p :\leftrightarrow \lambda x.$A ensures that the other predicates remain unchanged.

The language DESIRE [van Langevelde et al.,1993] uses the notion of meta-layered compositional architecture to specify a KBS. 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. The reasoning modules of DESIRE can be roughly identified with inference actions in $(ML)^2$ and KARL, but DESIRE provides much more sophisticated means to control the reasoning process of an inference action. An important distinction between DESIRE and the languages $(ML)^2$ and KARL is that DESIRE uses its object/meta-level distinction *to specify and to reason about* flexible control of object-level inferences whereas languages such as $(ML)^2$ and KARL define control of inferences by means of a procedural language. From a semantic point of view one difference between DESIRE (see [Treur, 1994]) and $(ML)^2$, KARL, and MCL lies in the fact that the former uses temporal logic with linear time for specifying the 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). Two different paths for computing the same input-output tuple are not distinguished. In DESIRE, the entire reasoning trace that leads to the derived output is used as the semantics. DESIRE uses a sequence of models $M_i,M_1,...,M_n,M_o$ to define the semantics of a specification. It therefore allows the expression of strategic reasoning about the history of the derivation process.

Transaction Logic [Bonner & Kifer, 1993] was developed to define a declarative semantics for state changes in logic programming and database updates. It also uses sequences of models as a semantics for database queries and updates. In a recent version [Bonner & Kifer, 1995] introduce two types of oracles. Oracles that inform about the truth values of a state and oracles that execute elementary state transitions. Transaction logic is used to construct composed transitions. The usual constructs of dynamic logic for specifying procedural control over the execution of transitions can be simulated in Transaction logic. In addition, *constraints* can be used to restrict possible derivation paths. Such a semantics that includes the derivation path by a sequence of models is also necessary when one wants to specify dynamic integrity constraints on the reasoning process which not only restrict relations between input and outputs but also define restrictions for the reasoning process itself.

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

Fig. 5 Algebraic specification of the choice operator in TFL.

The semantics of states and elementary state transitions remain outside the scope of Transaction logics. It is assumed to be provided by external oracles that can be seen as parameters of a specification in Transaction logic. For us, that would imply that the logical definition of the inference actions that define the elementary updates would remain outside the scope of the language that specifies control, as the case in KARL. Therefore, it covers only a part of our problem because we want to *integrate* the logical definition of inference actions as elementary transitions into a language expressing control over their execution. In contrast to Transaction logic, we integrate predefined elementary updates into the semantics and axiomatization of our approach. Still, Transaction logic is a very interesting point of reference when we extend our approach to a semantics based on model paths to express strategical reasoning.

6 Conclusions and Future Work

We analyzed existing approaches for specifying the reasoning process of KBSs. We derived general requirements for appropriate specification formalisms. The logic MCL which we presented generalizes the gist of solutions that were chosen by languages like KARL and $(ML)^2$ and provides an adequate mathematical framework for their uniform formalization. In a nutshell: our approach uses algebras to represent states of the reasoning process, bulk-updates that change algebras to express state transitions, and procedural constructs to define control over the execution of transitions. MCL defines a formal semantics and an axiomatic semantics for specification languages for KBSs. This formalization has several advantages compared to existing approaches in knowledge engineering:

- Different types of state transitions as provided by KARL and $(ML)^2$ are integrated. The datalog-like strategy of KARL that returns all answer substitutions and the Prolog-like strategy of $(ML)^2$ that returns one answer substitution are captured by two predefined state transitions operators.
- Problems like the nonintuitive interaction of logical and state-dependent variables and the use of interpreted logics are bypassed
- The formalization work on KBSs becomes comparable with related approaches in software engineering and database engineering. MCL can be used to formalize many of the existing approaches of other fields like evolving algebras and database update languages like PDDL and DDL.
- The axiomatization of MCL takes a step in the direction of automated proofs of reasoning processes of KBSs.

[Fensel et al., 1996] provide a case study using MLPM and extend it through the specification of goals (i.e., declarative specification of the desired functionality of a knowledge-based system). Proofs are provided that ensure that a specified reasoning strategy actually achieve the goals as required by a task definition. The Karlsruhe Interactive Verifier (KIV) [Reif, 1995] is applied to support semi-automatic proof support. KIV is based on dynamic logic and can be used to verify imperative programs against specifications in first-order logic. It represents the state of a reasoning process by value assignments of dynamic variables whereas MLPM and MCL apply the states as algebras approach. [Schönegge, 1995] provides an extension of KIV for a subset of evolving algebras (i.e., sequential and deterministic) that makes a step in overcoming this difference. A state is represented by the actual values of the functions of the signature. The main problem in immediately applying KIV to MCL specifications is that KIV is based on a point-wise modification of functions whereas MCL provides bulk-updates that modify a complete predicate in one step. This problem will disappear when KIV is extended to verify Evolving Algebras of parallel algorithms as this extension also requires bulk-updates.

Acknowledgment

We would like to thank Joeri Engelfriet, Pascal van Eick, Frank van Harmelen, Arno Schönegge, Yde Venema, Mark Willems, and two engaged anonymous reviewers for valuable comments and Andrew Butterfield and Jeff Butler for proof reading.

References

[Angele, 1993] J. Angele: *Operationalisierung des Models der Expertise mit KARL*, Infix, St. Augustin, 1993.

[Angele et al., 1993] J. Angele, D. Fensel, D. Landes, S. Neubert, and R. Studer: Model-Based and Incremental Knowledge Engineering: The MIKE Approach. In J. Cuena (ed.), *Knowledge Oriented Software Design, IFIP Transactions A-27*, North Holland, Amsterdam, 1993.

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

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

[Bonner & Kifer, 1995] A. J. Bonner and M. Kifer: *Transaction Logic Programming*, Technical Report CSRI-323, 1995.

[Börger, 1995] E. Börger: Why Use Evolving Algebras for Hardware and Software Engineering. In M. Bartosek et al. (eds.), *SOFSEM '95: Theory and Practice of Informatics*, LNCS 1012, Springer-Verlag, 1995.

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

[Brazier et al., 1995] F. Brazier, B. Dunin Keplicz, N. R. Jennings, and J. Treur: Formal Specification 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, 1995.

[Bylander, 1991] T. Bylander: Complexity Results for Planning. In *Proceedings of the 12th International Joint Conference on Artificial Intelligence (IJCAI-91)*, Sydney, Australia, August 1991.

[Bylander et al., 1991] T. Bylander, D. Allemang, M. C. Tanner, and J. R. Josephson: The Computational Complexity of Abduction, *Artificial Intelligence*, 49, pages 25—60, 1991.

[de Kleer, 1986] J. de Kleer: An Assumption-based TMS, *Artificial Intelligence*, 28, 1986.

[Feijs & Jonkers, 1992] L.M.G. Feijs and H.B.M. Jonkers: *Formal Specification and Design*, Cambridge Tracts in Theoretical Computer Science 35, 1992.

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

[Fensel, 1995b] D. Fensel: *The Knowledge Acquisition and Representation Language KARL*, Kluwer Academic Publ., Boston, 1995.

[Fensel, 1995c] D. Fensel: Formal Specification Languages in Knowledge and Software Engineering, *The Knowledge Engineering Review*, 10(4), 1995.

[Fensel, Angele & Studer] D. Fensel, J. Angele, R. Studer: The Knowledge Acquistion and Representation Language KARL, to apear in *IEEE Transactions on Knowledge and Data Engineering*.

[Fensel & Groenboom, 1996]D. Fensel und R. Groenboom: MLPM: Defining a Semantics and Axiomatization for Specifying the Reasoning Process of Knowledge-based Systems. In *Proceedings of the 12th European Conference on Artificial Intelligence (ECAI-96)*, Budapest, August 12-16, 1996.

[Fensel & Straatman, 1996]D. Fensel und R. Straatman: The Essence of Problem-Solving Methods: Making Assumptions for Efficiency Reasons. In N. Shadbolt et al. (eds.), *Advances in Knowledge Acquisition, Lecture Notes in Artificial Intelligence (LNAI)*, no 1076, Springer-Verlag, Berlin, 1996.

[Fensel & van Harmelen, 1994] D. Fensel and F. van Harmelen: A Comparison of Languages which Operationalize and Formalize KADS Models of Expertise, *The Knowledge Engineering Review*, 9(2), 1994.

[Fensel et al., 1996] D. Fensel, A. Schönegge, R. Groenboom and B. Wielinga: Specification and Verification of Knowledge-Based Systems. *Proceedings of the ECAI-96 Workshop on Validation, Verification and Refinement of Knowledge-Based Systems, 12th European Conference on Artificial Intelligence (ECAI-96)*, Budapest, August 12-16, 1996.

[Goldblatt, 1992] R. Goldblatt: *Logics of Time and Computation* (2nd edition), CSLI LectureNotes No. 7, Stanford, 1992.

[Goldblatt, 1982] R. Goldblatt: *Axiomatising the Logic of Computer Science*, LNCS 130, Springer-Verlag, Berlin, 1982.

[Groenboom, 1997] R. Groenboom: *Formalizing Knowledge Domains - Static and Dynamic Aspects*, PhD thesis, University of Groningen, Shaker Publ., 1997.

[Groenboom & Renardel de Lavalette, 1994] R. Groenboom and G.R. Renardel de Lavalette: Reasoning about Dynamic Features in Specification Languages. In D.J. Andrews et al. (eds.), *Proceedings of Workshop in Semantics of Specification Languages*, October

1993, Utrecht, Springer Verlag, Berlin, 1994.

[Groenboom & Renardel de Lavalette, 1995] R. Groenboom and G.R. Renardel de Lavalette: A formalization of Evolving Algebras. In *Proceedings of Accolade 95*, Dutch Graduate school in Logic, Amsterdam, 1995.

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

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

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

[Henkin, 1949] L. Henkin: The completeness of the first order functional calculus, *The Journal of Symbolic Logic*, 14:159—166, 1949.

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

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

[Kifer et al., 1995] M. Kifer, G. Lausen, and J. Wu: Logical Foundations of Object-Oriented and Frame-Based Languages, *Journal of the ACM*, vol 42, 1995.

[Koymans & Renardel de Lavalette, 1989] C.P.J. Koymans and G.R. Renardel de Lavalette: The logic MPL_{ω}, In M. Wirsing and J.A. Bergstra (eds.), *Algebraic Methods: Theory, tools and applications*, LNCS 394, Springer Verlag, 1989.

[Kripke, 1959] S.A. Kripke: A Completeness Theorem in Modal Logic, *Journal of Symbolic Logic,* 24:1—14, 1959.

[Lewis & Langford, 1932] C.I. Lewis and C.H. Langford: *Symbolic Logic*, The Century Co., 1932.

[van Langevelde et al.,1993] I. van Langevelde, A. Philipsen, and J. Treur: A Compositional Architecture for Simple Design Formally Specified in DESIRE. In [Treur & Wetter, 1993].

[Lloyd, 1987] J.W. Lloyd: *Foundations of Logic Programming, 2nd Editon*, Springer-Verlag, Berlin, 1987.

[McCarthy & Hayes, 1969] J. M. McCarthy and P. J. Hayes: Some Philosophical Problems from the Standpoint of Artificial Intelligence. In B. Meltzer et al. (eds.), *Machine Intelligence*, vol 4, Edinburgh University Press, 1969.

[Nebel, 1996] B. Nebel: Artificial intelligence: A Computational Perspective. In G. Brewka (ed.), *Principles of Knowledge Representation*, CSLI publications, Studies in Logic, Language and Information, Stanford, 1996.

[Pierret-Golbreich & Talon, 1996] C. Pierret-Golbreich and X. Talon: An Algebraic Specification of the Dynamic Behaviour of Knowledge-Based Systems, *The Knowledge Engineering Review*, 11(2), 1996.

[Pratt, 1976] V.R. Pratt: Semantical Considerations on Floyd-Hoare logic. In *Proceedings of the 17th annual IEEE Symposium on Foundations of Computer Science*, 1976.

[Przymusinski, 1988] T. C. Przymusinski: 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, 1988.

[Reif, 1995] W. Reif: The KIV Approach to Software Engineering. In M. Broy and S. Jähnichen (eds.): Methods, Languages, and Tools for the Construction of Correct Software, LNCS 1009, Springer-Verlag, Berlin, 1995.

[Renardel de Lavalette, 1984] G.R. Renardel de Lavalette: Descriptions in Mathematical Logic, *Studia Logica*, XLIII(3):281—294, 1984.

[Schönegge, 1995] A. Schönegge: Extending Dynamic Logic for Reasoning about Evolving Algebras, research report 49/95, Institut für Logik, Komplexität und Deduktionssysteme, University of Karlsruhe, 1995.

[Schreiber et al., 1993]A.T. Schreiber, B.J. Wielinga, and J. A. Breuker (eds.): *KADS: A Principled Approach to Knowledge-Based System Development, vol 11 of Knowledge-Based Systems Book Series*, Academic Press, London, 1993.

[Schreiber et al., 1994] A.T. Schreiber, B. Wielinga, J. M. Akkermans, W. Van De Velde, and R. de Hoog: CommonKADS. A Comprehensive Methodology for KBS Development, *IEEE Expert*, 9(6):28—37, 1994.

[Spee & in 't Veld, 1994] J. W. Spee and L. in 't Veld: The Semantics of K_{BS}SF, A Language For KBS Design, *Knowledge Acquisition*, 6(4), 1994.

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

[Spruit et al., 1993] P. A. Spruit, R. Wieringa, and J.-J. Meyer: Dynamic Database Logic: The First Order Case. In V.W. Lipeck and B. Thalheim (eds.), *Fourth International Workshop on Foundations of Models and Languages for Data and Objects*, Workshop in Computing, Springer-Verlag, Berlin, 1993.

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

[Treur, 1994] J. Treur: 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, Springer Verlag, LNCS 883, Berlin, 1994.

[Treur & Wetter, 1993] J. Treur and T. Wetter: *Formal Specification of Complex Reasoning Systems*, Ellis Horwood, New York, 1993.

[Ullman, 1988] J. D. Ullman: *Principles of Database and Knowledge-Base Systems, vol I*, Computer Sciences Press, Rockville, Maryland, 1988.

[Voss & Voss, 1993] H. Voss and A. Voss: 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, 1993.

[Weiß, 1995] G. Weiß: Adaption and Learning in Multi-Agent Systems: Some Remarks on a Bibliography. In G. Weiß et al. (eds.), Adaption and Learning in Multi-Agent Systems, Lecture Notes in Artificial Intelligence (LNAI), no 1042, Springer-Verlag, 1995.

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