In Proceedings of the Workshop on Information systems - Correctness and Reusability (IS-CORE), Amsterdam, September 27-30, 1994.

The Specification Language KARL and Its Declarative Semantics

Dieter Fensel, Jürgen Angele, and Rudi Studer
Institut AIFB, University of Karlsruhe
76128 Karlsruhe, Germany
phone: 49-721-6084754, fax: 49-721-693717
e-mail: fensel@aifb.uni-karlsruhe.de

Abstract. The *Knowledge Acquisition and Representation Language (KARL)* combines a description of a knowledge-based system (kbs) at the conceptual level (a so-called *model of expertise*) with a description at a formal and executable level. It is a specification language which allows the precise and unique description of a kbs independently from implementational details. In the paper, KARL is mainly discussed as a formal language. That is, the paper introduces a *formal semantics* for KARL. Because KARL allows the representation of static and dynamic (i.e., procedural) knowledge, its semantics must integrate both types of knowledge. First, an object-oriented logic L-KARL was developed which can be used to specify static knowledge. Second, dynamic logic was used to develop P-KARL for specifying knowledge about dynamics. Third, both languages had to be combined to represent a complete model of expertise. As a result, the integrated description of static and dynamic knowledge based on a well-defined declarative framework becomes possible.

Introduction

The development of the language KARL is part of the *MIKE-approach* (*Model-based and Incremental Knowledge Engineering*) [AFL+93], which aims at a development method for knowledge-based systems (kbs) covering all steps from initial knowledge acquisition to design and implementation. In the past, most expert systems or kbs were developed according to the rapid prototyping approach. The acquired knowledge was immediately implemented and the running prototype was used to guide the knowledge acquisition process. The distinction of symbol level and knowledge level [New82] created the conceptual framework for different process models for the development of kbs. A knowledge level description of the *task* solved by the system and the *knowledge*, which is required to solve the task, is constructed during a modelling activity. This knowledge level description is built independently of design and implementation of a system. The separation of analysis and design/implementation resembles a lesson learnt in software engineering in the late sixties in reaction to the so-called software crisis.

Meanwhile, informal and semiformal description techniques like the KADS *model of expertise* [SWB93] have been developed for specifying the knowledge independent from its implementation. These techniques fulfil the same purpose for kbs-development as dataflow diagrams, entity-relationship diagrams, state-transition diagrams, etc. in software engineering and information system development.

Informal techniques enable the specification of a system at a conceptual level in an easy and understandable manner. Otherwise, it is well-known from software engineering that software specifications using these description techniques have some well-known shortcomings. Informal or semiformal specifications using natural language suffer from ambiguity and impreciseness and can neither be evaluated by automatic procedures nor by formal proofs. Therefore, the *formal knowledge specification languages* DESIRE [LPT93], FORKADS [Wet90], K_{BS}SF [JoS92], $(ML)^2$ [HaB92], MODEL-K [KaV93], MoMo [VoV93], OMOS [Lin93], QIL [ARS92], and KARL have been developed to improve the result of the specification phase by supplementing informal descriptions.[1] A formal description reduces the vagueness and ambiguity of natural-language descriptions by defining a precise and detailed semantics. Such a formalized description allows formal proofs of properties of the specified knowledge. Furthermore, some of these languages provide an automatic mapping on an (inefficient) operational description to permit prototyping as a means for knowledge evaluation. The main concerns of the *Knowledge Acquisition and Representation Language (KARL)* which is one of these languages are:

- to provide appropriate modelling primitives (allowing knowledge specifications at the so-called *knowledge level*): KARL distinguishes several types of knowledge and defines different language primitives for them. Most of these primitives have a *graphic* representation to support the usefulness of KARL specifications as a medium of communication.
- to provide *formality*: KARL has a declarative (i.e., model-theoretic, semantics). This semantics must allow the representation of static and dynamic (i.e., procedural) knowledge.
- to provide *executability*: KARL´s declarative semantics has been operationalized and a debugger has been implemented.

In the paper, we will focus on the declarative semantics of KARL which is its backbone. This semantics defines precisely the meaning of every expression. In addition, it serves as base line for developing formal proof procedures and an operationalization which allows (inefficient) prototyping. The contents of the paper are organized as follows. First, the language KARL is sketched in section one. Then the semantics of the sublanguages L-KARL and P-KARL and finally the semantics of the complete language KARL are given in sections two, three, and four. Because of the limited space, the semantics of L-KARL is sketched only briefly. A complete definition of the semantics and of the language can be found in [Fen93]. We do not only describe the semantics but also give the main reasons for our choices. The declarative semantics of KARL has been developed with an eye to its operationalization. Section five discusses how the mechanizability influenced design decisions of the declarative semantics. Section six sketches the tool environment of KARL and gives some of its applications. Section seven compares KARL to some other specification languages.

1 The Language KARL

In the following, we first introduce the conceptual model which underlies a KARL specification. Then, the two sublanguages of KARL are sketched. The sublanguage Logical-KARL (L-KARL) integrates frames and logic to specify static knowledge. The sublanguage Procedural-KARL (P-KARL) is used to specify the control flow of a problem-solving process.

1. A description of most of these languages and their comparison can be found in [FeH94].

1.1 The Model of Expertise

The conceptual model underlying KARL is derived from the KADS *model of expertise* [SWB93]. As KADS is the most prominent methodological approach to expert system development—at least in Europe—we tried to keep the KARL model as close as possible to the KADS model of expertise. Otherwise, this model is defined informally only and refinement and modifications are natural results of every formalization process. For the sake of self containment we sketch briefly the KADS model of expertise before we introduce how this model was realised and modified by the KARL model of expertise.

The KADS Model of Expertise

A very important part of the KADS methodology [SWB93] is the *model of expertise* which describes the different kinds of knowledge required to solve the given tasks. The model of expertise distinguishes different types of knowledge, defines primitives to express them, and organizes them into several layers. Precisely it distinguishes static knowledge and three types of control knowledge.[2] The goal of a model of expertise is to provide a model of the problem solving behaviour independently of a certain implementation.

Domain layer: It represents knowledge about the application domain of the system. An important property of the domain layer is that the knowledge should be represented as independently as possible from the way it will be used. It has two main purposes. First, it should define a conceptualization of the domain. Secondly, it should define a declarative theory of the domain providing all the domain knowledge required to solve the given task.

Inference layer: This second layer defines the first type of control knowledge. It specifies the inferences that constitute a problem-solving method and specifies how to *use* the knowledge from the domain layer in these inferences. This is done in two ways: the inference layer specifies

- the *inference steps* that can be made using the domain knowledge, and
- the *knowledge roles,* which model the premises and conclusions of the inferences.

The inference steps are assumed to be elementary in the sense that they are completely described by their names, an input/output specification and a reference to the domain knowledge that they use. The inference layer specifies the inference steps and knowledge roles as well as the data-dependencies between these steps and roles. These dependencies are specified in a network of inference actions and knowledge roles known as an *inference structure*. The inference layer *restricts* the use of the domain layer knowledge and *abstracts* from it. It restricts all possible inferences to the set of inferences which are defined by it. This is done to improve the efficiency of the problem-solving process. The inference layer abstracts from the domain layer by using task-specific names for inferences and roles. The domain-independent formulation of the inference layer should support its reuse, i.e. its application for similar tasks in different application domains.

A *domain view* must specify the relationship between the generic terms used at the inference layer and the domain-specific knowledge specified at the domain layer. Mainly, roles have to be connected with domain classes and inference actions have to be connected with knowledge required for such an inference.

Task layer: A task represents a fixed strategy for achieving problem solving goals. The purpose of the task layer is to specify *control* over the execution of the basic inference steps specified at the inference layer. This is done by imposing an ordering on these steps in terms of execution sequences, iterations, conditional statements etc. The description of a task consists of three components: The goal which is fulfilled by the task; the control terms which correspond to knowledge roles of the inference layer and which are used to specify conditions for the control flow; and the task structure which hierarchically refines a given task to subtasks and elementary steps, i.e. inference actions.

The Refined Model of Expertise of KARL

Originally, KADS proposed KL-ONE as language for the domain layer. KL-ONE defines a very restricted set of language primitives which enables strong characterisations of decidability and efficiency of reasoning with it. Yet, for a specification language a broad syntactical variety of modelling primitives seems necessary to make the step from an informal to a formal description as smooth as possible. Therefore, KARL integrates concepts of object-oriented databases and logic for the domain layer, the inference layer, and their connections. KARL provides the sublanguage *Logical-KARL (L-KARL)* for this purpose. L-KARL is derived from Frame-logic (F-logic) [KLW93]. Terminological knowledge can be described by a taxonomy of classes. For each class, attributes can be defined and are inherited according to the taxonomy. Further knowledge can be described with logical formulae. A domain layer is structured and hierarchically ordered by the is-a hierarchy between classes and a module hierarchy.

Besides its use at the domain layer, L-KARL is used to specify the logical relationship defined by an inference

2. Because there is still significant disagreement about the third type of control knowledge (i.e., the *strategic layer*) we have neither regarded it for the KARL model nor will we further discuss it in this paper.

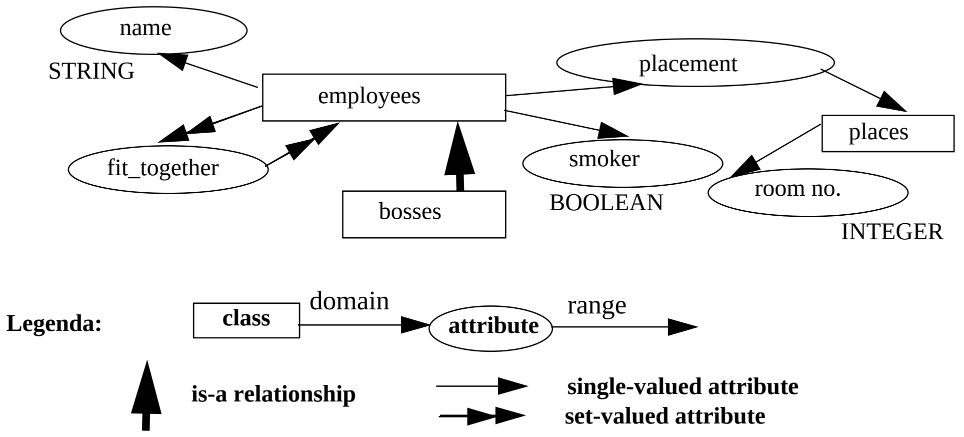

Fig. 1 The domain layer of the Sisyphus example.

action at the inference layer. Extending KADS, L-KARL can be used to define a terminological structure of a knowledge role. In KADS, such roles are flat containers, whereas in KARL they can be used to define a *problem-solving-method-specific terminology* independently from the domain-specific terminology. The need for such a terminology is one of the most significant results of the role-limiting method approach ([Mar88], [Pup93]). A second improvement compared to KADS at the inference layer is the introduction of *hierarchical refinement* similar to levelled dataflow diagrams [You89]. Therefore, in KARL large specifications are manageable.

Furthermore, L-KARL is used to specify the *domain view*. Modified Horn logic can be used to define a view from the problem-solving method on the domain knowledge. Such a *view* could be used to provide domain knowledge for an inference action at the inference layer. Complementary, a *terminator* could be used to write results of a problem-solving process at the inference layer back to the domain layer and therefore to re-express such a result in domain specific terms.

Procedural knowledge: The sublanguage *Procedural-KARL (P-KARL)* is used to specify the control flow of a problem-solving method at the task layer. Sequence, branch, loop, and procedure call are the means to specify control flow and the hierarchical task structure. Conditions can be specified via logical statements about the contents of knowledge roles. The goal of a task is described informally only.

The Sisyphus Example

We apply the so-called Sisyphus-I example to illustrate the different language primitives of KARL. Sisyphus is a project that aims at comparing different approaches to aspects of knowledge engineering [Lin92]. An assignment problem was posed, in which employees are assigned to office places with several requirements to be met. An example for the graphical representation of the domain layer of the model of expertise is given in Figure 1. The domain terminology and the domain knowledge required for the problem-solving method is defined at the domain layer. It consists of three classes. A class of employees which should be placed and a set of working places for them. A third class defines the subset of all employees which are bosses. Employees are described by their names, whether they smoke, and by the set of other employees with whom they can share a room. The places are described by their room numbers. The attribute *placement* should represent the solution, that is, for each employee his place. The inference layer as shown in Figure 2 contains the elementary inference steps and knowledge roles of the problem-solving method. Components (employees) and Slots (places) are combined by the inference action *Create*. *Prune* eliminates illegal states by using the domain knowledge delivered by the view *correct*. *Check* searches for valid solutions which are saved via the terminator *solution*. The control flow between these inferences is defined at the task layer (see Figure 3). It consists of a loop which determines when a solution has been found. This simple control flow does not regard the case that no solution exists but the example should be kept as simple as possible.

1.2 Logical-KARL (L-KARL)

L-KARL is a customization of Frame-logic (F-logic) [KLW93]. F-logic and L-KARL enrich the modelling primitives of first-order logic by syntactic modifications but preserve the model-theoretical semantics of it. In this way, ideas of semantical and object-oriented data models are integrated into a logical framework enabling the declarative description of terminological as well as assertional knowledge.

L-KARL distinguishes classes, elements, and values. *Classes* refer to sets of real-world objects with common features. Classes and elements can be described by attributes values. Attribute and class definitions together with

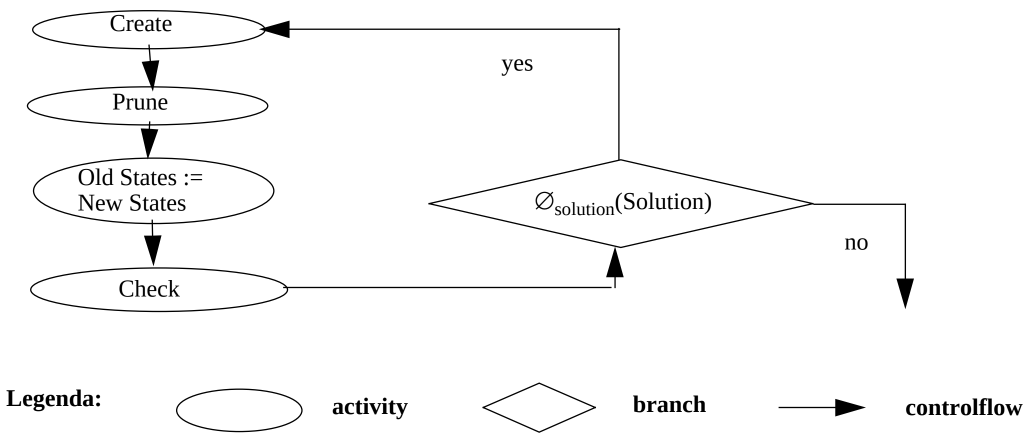

Fig. 3. The task layer of the Sisyphus example.

an is-a hierarchy and multiple attribute inheritance can be applied to describe *terminological knowledge*. These attributes have defined domain and range and can be applied to describe elements of classes or to the classes itself. *Elements* refer to real-world objects whereas *values* are only used to describe such objects. *Intensional* and *factual knowledge* is described by logical relationships between classes, objects, and their values. In the following, we will discuss the different modelling primitives.

Elements are denoted by *element-id-terms*, consisting of variables, functions, or element-constants, similar to terms in first-order logic. By means of functions it is possible to generate new object identifiers in logical expressions. This way to generate new objects is based on O- and F-Logic (cf. [KLW90, KiW93]). Classes are denoted by *class-id-terms*, which consist of variables or class constants. The class constants are the class names of the class definitions. A *value-id term* denotes a value, i.e. a number, a boolean value, or a string.

A *class definition* which corresponds to a frame describes class attributes which refer to the class as such and attributes for the objects which are elements of the class. The attributes are described by their name, their domain, and their range. Classes are arranged in an is-a hierarchy with multiple attribute inheritance. Attributes can be single-valued or set-valued. As already mentioned, attributes can be used to describe elements as well as classes. They have defined domain types and range types. In the general case, the range is defined by a set of classes. A correct attribute value must be an element or a subclass of all classes used in its range definition. The specification of attributes via class definitions therefore defines the following well-typing conditions:

- First, there must be a functional dependency between an object and the values of its attributes.
- Second, an attribute can only be applied to a class or element for which it is defined by an according class definition.
- Third, the values of attributes must fulfil the range restriction of the corresponding class definition.

These well-typing conditions are integrated into the model-theoretical semantics of L-KARL [Fen93]. That is, a

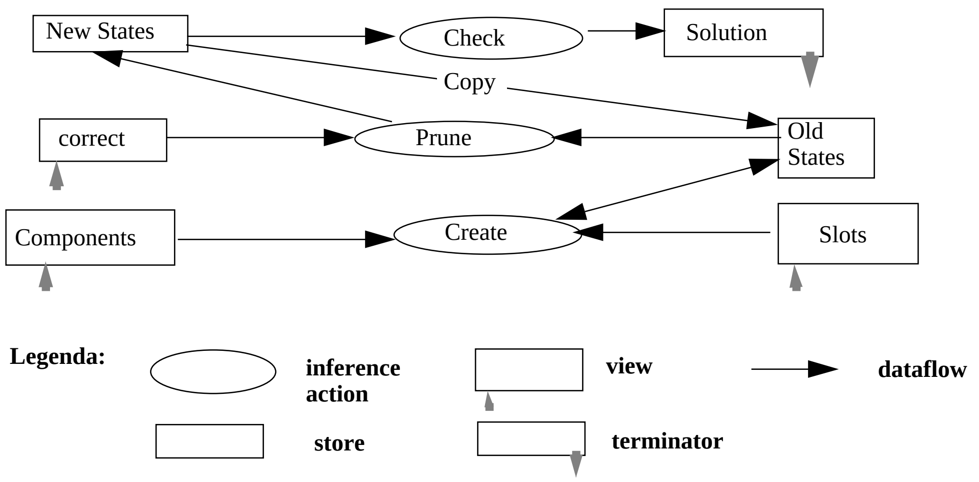

Fig. 2. The inference layer of the Sisyphus example.

minimal Herbrand model is only regarded as valid semantics when it fulfils these restrictions (see section two).

The literals of logical expressions in L-KARL are *is-element-of literals* which describe that objects are elements of classes; *is-a literals* which describe sub- and super set relationships between the classes; *equality literals* which describe equality of objects, classes, and values; and finally *data literals* which define attribute values for objects and classes. Logical formulae are built from these literals using logical connectors $\wedge$, $\vee$, $\neg$, $\leftarrow$ and variable quantification. The logical language to describe relationships between classes, objects, and values is Horn logic with equality and function symbols extended by stratified negation (cf. [Prz88], [Ull88]).

Def. 1 (Positive Literal in L-KARL)[3]

A positive literal in L-KARL is either:

- an *is-element-of literal* $c \in c$ where e is an element-id-term, c is a class-id-term. An element-term describes that an object e is an element of class c.
- an *is-a literal* $c \leq c'$ where c and c' are class-id-terms. An is-a-term expresses that a class c is a subclass of class c'.
- a *data literal* $o[..., a{:}T,..., s{::}\{S_1,...,S_n\},...]$ where o is either an element-id-term or a class-id-term, T, S_i are again data-literal. a is an attribute name of a single-valued attribute, s of a set-valued attribute. A data-literal defines attribute values for the element or class o.
- an *equality literal* $o = o'$ where o and o' are id-terms. This means that o and o' denote the same element, class, or value.

In addition, *P-literals* $p(a_1{:}T_1,..., a_n{:}T_n)$ allow to express relationships between data literals T_i in a similar way as in predicate logic. In extension, the arguments of a predicate are named and typed.

Figure 1 specifies the terminological knowledge of the domain layer of our running example in the graphical notation. In addition, intensional and factual knowledge have to be defined. Some employees together with their names and some places have to be given. . Attributes can be marked as input or output attributes. Input attributes

class: employees

Element-id	*name*	*fit_together*	*placement*	*smoker*
fvh	Frank van Harmelen			NO
mab	Manfred Aben			NO
dla	Dieter Landes			NO
sun	Susanne Neubert			YES
jtr	Jan Treur			NO

class: boss

Element-id	*name*	*fit_together*	*placement*	*smoker*
jtr	Jan Treur			NO

refer to case-specific input of the user, describing his specific problem. Output attributes store the results of a problem-solving process. For the reason of space limitation, we will not discuss this distinction further on (see [Fen93] for more details). In our example there are no values given for the attribute *placement* because this attribute will contain the solution of the problem-solving process. The attribute *fit_together* is defined by means of logical clauses. The rows in the pictured tables correspond to ground clauses like:

fvh [*name* : *“Frank van Harmelen“*, *smoker*: NO] ∧ *fvh* ∈ *employees*

which can be short-cut by a combined is-element-of term and data term

fvh [*name* : *“Frank van Harmelen“*, *smoker*: NO] ∈ *employees*

Two employees fit together if nobody of them is a boss and if both smoke or none of both smokes:

$$\begin{aligned} &\forall X\, \forall Y\, \forall Z_1\, \forall Z_2\, (X[\text{fit\_together} :: \{Y\}] \leftarrow \\ &\qquad X[\text{smoker} : Z_1] \in \text{employees} \wedge \\ &\qquad Y[\text{smoker} : Z_2] \in \text{employees} \wedge \\ &\qquad \neg(X \in \text{bosses}) \wedge \neg(Y \in \text{bosses}) \wedge \\ &\qquad Z_1 = Z_2). \end{aligned}$$

It is easy to see that the elements *jtr* and *sun* do not fit to any other element whereas *fvh*, *mab*, and *dla* fit altogether. L-KARL is also used to specify the logical inferences at the inference layer and their connection with the domain

3. In honour to F-logic we call these literals also *F-literals*.

layer. We will partly show this. In fact, we give the definition of the inference action *Prune,* its input store *Old States*, its output store *New States*, and its view *correct* in Figure 4. One remark should be made. The stores *New States* and *Old States* seem to be redundant because they specify both the same terminology. Both specify that a state (new respectively old) consist of a set of assignments and each assignment is a pair of a component and a slot. Otherwise, the distinction between two different stores is the way how the implicit non-monotonicity of the inference action *Prune* is represented. *Prune* should delete states which describe incorrect assignments. As deletion of elements cannot be expressed immediately in a declarative manner, the inference action Prune describes that old states which are not wrong are also regarded as new states. The old states become then overwritten by the new states via an assignment expressed at the task layer (cf. Figure 3). The non-monotonic parts of a specification are shown at this layer.

The inference action is very easily to specify as it is shown in Figure 4. Every old state which is not a wrong state is a new state. That is, such a state is investigated further on during the problem-solving process until all components have been assigned to a slot. In order to decide whether a state is correct or wrong domain knowledge is required by the inference action. The required domain knowledge is delivered by the view *correct*. It specifies when a state is a wrong state. A state Z having two assignments A_1 and A_2 where different employees are assigned to the same room is a wrong state if the assigned employees do not fit together. The set-valued attribute *fit_together* was defined at the domain layer.

1.3 Procedural-KARL (P-KARL)

Experiences in XCON (cf. [SBJ87], [BaS89]) showed that great problems arise if control flow is specified implicitly only. A production rule formalism was used to specify a large expert system for designing computer configurations. Very soon it became clear that the domain experts have a significant amount of control knowledge concerning the appropriate order of sub activities and that this knowledge is required to solve a task efficiently. Therefore, this knowledge was implicitly encoded into the rule formalism. Similar experiences have been made

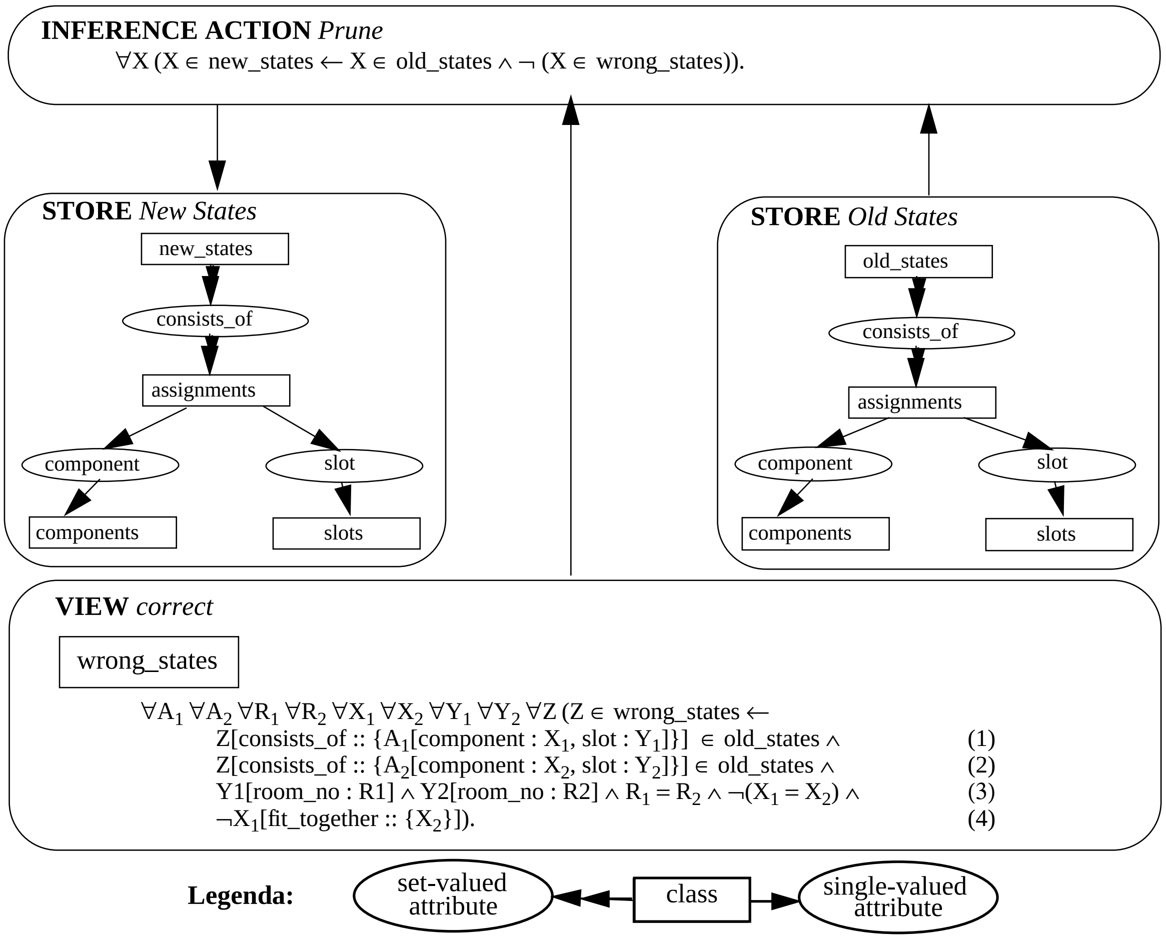
INFERENCE ACTION *Prune*
$\forall X (X \in \text{new\_states} \leftarrow X \in \text{old\_states} \wedge \neg (X \in \text{wrong\_states}))$.
STORE *New States*
new_states
consists_of
assignments
component
slot
components
slots
STORE *Old States*
old_states
consists_of
assignments
component
slot
components
slots
VIEW *correct*
wrong_states
$\forall A_1 \forall A_2 \forall R_1 \forall R_2 \forall X_1 \forall X_2 \forall Y_1 \forall Y_2 \forall Z (Z \in \text{wrong\_states} \leftarrow$
$Z[\text{consists\_of} :: \{A_1[\text{component} : X_1, \text{slot} : Y_1]\}] \in \text{old\_states} \wedge$ (1)
$Z[\text{consists\_of} :: \{A_2[\text{component} : X_2, \text{slot} : Y_2]\}] \in \text{old\_states} \wedge$ (2)
$Y1[\text{room\_no} : R1] \wedge Y2[\text{room\_no} : R2] \wedge R_1 = R_2 \wedge \neg(X_1 = X_2) \wedge$ (3)
$\neg X_1[\text{fit\_together} :: \{X_2\}])$. (4)
Legenda:
set-valued attribute
class
single-valued attribute

Fig. 4. The inference action *Prune* and its context.

```
WHILE $\varnothing_{solution}$(Solution)
DO
        Old States := Create (Old States);
        New States := Prune (Old States);
        Old States := New States;
        Check (New States);
ENDDO
```

Fig. 5. The task layer of the Sisyphus example defined by using the linear notation of P-KARL.

with PROLOG where the order of clauses and literals and primitives like the cut are used to implicitly specify control flow. This need for specification of control flow leads to the development of the sublanguage P-KARL. The control flow is specified similar to procedural programming languages.

A P-KARL program consists of elementary programs which are assignments. Composed programs are built up from elementary ones by defining the control flow between these assignments. For this purpose, logical expressions can be used for defining conditions for branches and loops.

For the *elementary programs*, a number of function symbols $F = \{f_1, f_2,..., f_r\}$ and a number of variables $\{X_1,..., X_n\}$ are available. The function symbols correspond to names of inference actions. The variables address their stores and terminators.[4] The actual parameters of a function are the input stores of the corresponding inference action and the results of the function are mapped to its output stores. As primitive programs three types of *assignments* exist:

- (1) $(X_{k1},..., X_{kh}) := f_i(X_{j1},..., X_{jl})$
  f_i corresponds to an inference action and the X_{ks} denote its output stores and terminators and the X_{js} its input stores;
- (2) $X_k := X_i$
  X_k and X_i denote stores;
- (3) $b_j := b_i$
  b_j and b_i are logical variables.

Formulae are defined as follows:

- *true* and *false* are formulae;
- b is a formula if b is a logical variable;
- $\varnothing_c(X)$, where X is a store or terminator name and c is a class name, is a formula[5];
- and if ϕ, γ are formulae then $\neg\phi$, $\phi \vee \gamma$, $\phi \wedge \gamma$ are formulae.
- These are all formulae.

Formulae like $\varnothing_c(X)$ can be used to check the content of a store during the problem-solving process. In addition boolean variables can be defined for storing truth values which can then be used in other formulae.

A *composed program* is defined as

- *sequence*: *p;q*,
- *while loop*: *WHILE* ψ *DO p ENDDO*,
- *repeat-until loop*: *REPEAT p UNTIL* ψ, or
- *alternative*: *IF* ψ *THEN p ELSE q ENDIF*

of programs *p*, *q* and formulae ψ.

In our Sisyphus example, we have
  F = {*Check*, *Create*, *Prune*}
  X = {*New States*, *Old States*, *Solutions*}

The control flow consists of a loop of creating states, pruning them, and checking whether a solution has been found (see Figure 3). The linear notation is given in Figure 4.

The syntax of P-KARL is just sugar on top of dynamic logic (cf. [Har84], [Koz90]) which defines the gist of the language. In section three we will show how dynamic logic is used to define a declarative semantics for P-KARL. We chose dynamic logic because it is a well-known means to declaratively describe procedural programs.

4. Views are used to read domain knowledge and have no dynamic interpretation. Therefore, they do not appear at the task layer.

5. The formula $\varnothing_c(X)$ is evaluated true for a state if the store or terminator X does not contain an element of the class c.

2 The Declarative Semantics of Logical-KARL (L-KARL)

Because L-KARL is a syntactical variant of first-order predicate logic its semantics can be defined by modifying and extending the model-theoretical semantics of predicate logic. At first, an interpretation for a set of L-KARL formulae has to be defined. Objects, classes, and values are interpreted by individuals of a given *universe*. Associating classes and also sets (sets can appear as attribute values) with individuals allows to reason about classes and sets without destroying the first-order semantics of the language (cf. [KLW93]). For example, the following Horn clause collects all subclasses of class *c* as value of the set-valued attribute *subclass*:

$\forall X\ (c[subclass :: \{X\}] \leftarrow X \le c)$.

The attributes and their domain and range restrictions are interpreted by *functions* on the universe. Specific conditions postulated for these functions realize multiple-attribute inheritance. The is-a and is-element-of hierarchy are captured by a *partial ordering* defined on the universe.

As L-KARL is a syntactical extension of first-order logic it is necessary to define its semantics in a more complex manner.

Def. 2 An interpretation of an L-KARL language

An interpretation *I* for a L-KARL language is a tuple $<U_E, U_\Sigma, U_V, \le_U, I_U, I_{E\rightarrow}, I_{\Sigma\rightarrow}, I_{E\rightarrow\rightarrow}, I_{\Sigma\rightarrow\rightarrow}, I_{E\Rightarrow}, I_{\Sigma\Rightarrow}, I_{E\Rightarrow\Rightarrow}, I_{\Sigma\Rightarrow\Rightarrow}, I_\Pi, I_{A\Pi}>$ with:

- U_E is a subset of the domain to interpret element denotations, U_Σ is a subset of the domain to interpret class denotations, and U_V is a subset of the domain to interpret values. The sets must be disjunct and their union *U* defines the *domain*.
- $\le_U$ is a partial ordering on *U* used to interpret is-a and is-element-of literals.
- I_U interprets every element denotation, class denotation, and value by an element of the domain.
- $I_{E\rightarrow}$ interprets each single-valued attribute *a* defined for elements as a partial function

  $I_{E\rightarrow}(a)\colon U_E \rightarrow U.$
- $I_{E\rightarrow\rightarrow}$ interprets each set-valued attribute *s* defined for elements as a partial function

  $I_{E\rightarrow\rightarrow}(s)\colon U_E \rightarrow \wp(U).$

  $I_{\Sigma\rightarrow}$ and $I_{\Sigma\rightarrow\rightarrow}$ do the same for attributes defined for classes.
- $I_{E\Rightarrow}$, (and respectively $I_{\Sigma\Rightarrow}$, $I_{E\Rightarrow\Rightarrow}$, $I_{\Sigma\Rightarrow\Rightarrow}$) interprets each single-valued attribute *a* defined for elements as a *partial anti-monotonic function* having the *upwardly-closed subsets* of $\wp(U_\Sigma)$ as its range.
- I_Π interprets each *k*-ary predicate symbol by a *k*-ary relation over the domain and $I_{A\Pi}$ interprets each *k*-ary predicate symbol by its argument types.

$I_{E\rightarrow}$, $I_{E\rightarrow\rightarrow}$, $I_{\Sigma\rightarrow}$, and $I_{\Sigma\rightarrow\rightarrow}$ interpret each attribute using a partial *function* to capture the functional dependencies between objects and their attribute values. They interpret each attribute using a *partial* function as an attribute need not be defined for all elements or classes. The partiality of the function is used to interpret the *domain restrictions* of the attributes. The *range restriction* are handled by $I_{E\Rightarrow}$, $I_{\Sigma\Rightarrow}$, $I_{E\Rightarrow\Rightarrow}$, and $I_{\Sigma\Rightarrow\Rightarrow}$ as proposed by [KLW93]. A partial function *f* on *U* is *anti-monotonic* if the facts that $v \le_U u$ and $f(u)$ are defined imply that

(1) $f(v)$ is defined and

(2) $f(u) \subseteq f(v)$.

(1) ensures attribute inheritance. That is, if a class is a subclass of a class or if an element is an element of a class it inherits its attributes. (2) ensures that the range restriction for an inherited attribute can be stronger for a subclass, that is, its definition contains more classes of which an attribute value has to be an element or a subclass. This can occur if a class has several super classes defining the same attribute. The subclass inherits all range restrictions of its super classes. Therefore, the concept of an anti-monotonic function captures multiple-attribute inheritance in a declarative manner. *Upwardly-closed sets* are used as an interpretation for range restrictions because it should be ensured that if *c* is used as a range restriction and $c \le_U c'$ then c' must also hold true as a range restriction, because *c* is a subclass of c'.

Such an interpretation *I* defines a *model* for a set of formulae if every formula is true according to the interpretation. Elementary formulae are *is-element-of literals, is-a literals*, *equality literals*, and *data literals*. Therefore, it must be defined when an interpretation satisfies such a literal. Composed formulae which are built up from these elementary formulae by logical connectives and quantifiers have the usual interpretation. For example, a ground data literal

fvh[*name* : "*Frank van Harmelen*", *fit_together* :: {*mab, dla*[*name* : "*Dieter Landes*"]}]

is satisfied by an interpretation *I* iff:

- The functions which interpret the attributes *name* and *fit_together* are defined for *fvh*. That is, the domain restrictions for the attributes are not violated.

- The value of the function which interprets the attribute *name* is equal to *“Frank van Harmelen”*. That is, the functionality of an attribute is not violated and *“Frank van Harmelen”* does not violate the range restriction of it.
- The value of the function which interprets the attribute *fit_together* is a super set of the set *{mab, dla}*. It is not required, that a data literal contains all elements of a set-valued attribute.
- As the data literal recursively contains a second data literal *dla[name : “Dieter Landes“]*, it is required that this data literal is fulfilled by *I*, too.

An interpretation which fulfils a set of clauses is called a *model* of this set of clauses. Instead of all models, only a specific type of models is taken into account for L-KARL. In fact, only *Herbrand models* are admitted. This does not cause a significant restriction because every set of formulae can be transformed into a logical equivalent set of clauses and for every set of clauses *S* holds:

S has a model iff *S* has a Herbrand model [Llo87].

A Herbrand model *H* is a set of ground positive literals. These literals are regarded to be true according to the given Herbrand model *H*. A negative literal is true if the corresponding positive literal is not an element of *H*. Similar to predicate logic with equality, it is required that such a Herbrand model is *closed with regard to logical entailment.* That is, if a ground positive literal ϕ is entailed by *H* then it must hold: $\phi \in H$. A *congruence relation* on the Herbrand universe must be defined where each congruence class contains all syntactical different but semantical equivalent terms (see [SpA91] for more details).

In the case of Horn clauses, every set of clauses has a definite minimal Herbrand model. This *minimal* or *least Herbrand model* is taken as semantics for a set of L-KARL Horn clauses *S*. It is exactly the set of positive ground literals entailed by *S*. Negative literals are derived applying the *closed-world assumption*. That is, every negated ground positive literal $\neg\phi$ is regarded as being true iff $\phi \notin H$. Therefore, the set of negative literals which are entailed from a set of Horn clauses according to the minimal Herbrand model is larger than the set of negative literals which are entailed from all models. Because Horn clauses are too restrictive, L-KARL allows also negative literals as premises of implications. In this case, the minimal Herbrand model is no longer unique. That is, several minimal models can exist. Therefore, only a stratified set of clauses is allowed and *stratification* is applied to select one unique model from the set of all minimal Herbrand models. This model is called the *perfect Herbrand model* (cf. [Prz88], [Ull88]). Regarding only one model (i.e., the perfect Herbrand model) instead of all possible models makes the operationalization of KARL significantly easier.

3 The Declarative Semantics of Procedural-KARL (P-KARL)

One design goal of KARL was to give it a declarative character. That is, to define a model-theoretical semantics for it. For defining a declarative semantics for the procedural part we use *dynamic logic* (cf. [Har84], [Koz90]) which is a means for defining a model-theoretical semantics for procedural programs. The basic ingredients of dynamic logic are:

- *States* and *variables*: a state of a program is characterized by the values which are assigned to all its variables.
- *Programs*: a program is a binary relation between states.
- *Formulae*: A formula is true for some states and false for the others. That is, it is interpreted by a subset of all possible states for which it is regarded as true.

According to the expressive power of these formulae, different types of dynamic logic, e.g., propositional and first-order dynamic logic, can be distinguished. P-KARL restricts first-order dynamic logic to *deterministic while programs* [Koz90]: First, P-KARL allows only formulae without quantifiers (i.e., the alphabet of P-KARL does not contain quantifiers). This ensures that P-KARL programs are *regular* [Koz90]. Second, KARL allows only deterministic programs. That is, a program corresponds to a partial function between input states and output states. On the one hand, both language design decisions support the operationalization of KARL. On the other hand, it restricts the expressibility of the language. Till now, these restrictions have not caused problems but it is clear that the design of our language immediately biased our point of view on how to model a problem.

An *interpretation* $(D, F^A = \{f_1^A,..., f_r^A\}, P^A = \{P_1^A,..., P_s^A\})$ of a dynamic logic language using the multiple world semantics consists of three sets. First, a *domain* or *universe D* has to be defined. This domain defines the values that are assigned to the variables. Second, a set of functions F^A is required which serve as interpretations for the function symbols used in elementary programs. Third, a set of relations P^A has to be provided which serve to interpret the predicate symbols used in the formulae. In the next section is shown, how a L-KARL language is used to define such an interpretation.

4 The Declarative Semantics of KARL

In the following, the local semantics of an elementary inference action as well as its use to define a global semantics of a complete KARL specification is given.

4.1 The Semantics of an Inference Action

The declarative semantics of an elementary inference action is the perfect Herbrand model of the clauses and facts which describe it. In the case that it has an input store, its semantics implicitly relies on the dynamic state of the reasoning process as the facts of the input store must be contained by this perfect Herbrand model.

Def. 3 Semantics of an elementary inference action

The semantics of an elementary inference action is defined as the *perfect Herbrand model* of the union of

- the set of clauses and facts which define the inference action;
- all clauses and facts which define the views of it,
- all clauses and facts which define the terminators of it; and
- the *current* contents (i.e., sets of ground facts) of the input stores of it.
- In addition, if the inference action has at least one view or terminator then the clauses and facts defined at the domain layer are added.[6]

For example, the semantics of the inference action *Prune* is the perfect Herbrand model of the current contents of the store *Old States*, the clauses and facts used to define the domain layer, the view *correct*, and the inference action. If such a model does not exist or if it is not well-typed, the inference action is not defined for the given input.

Having defined the semantics of an inference action it remains to define *how such an inference action changes the contents of its output stores*.[7] For this purpose a subset of the perfect Herbrand model *H* is used. Every store is described by a set of class definitions which are used to define these subsets. So, a store can be used to select just this information out of the entire Herbrand model of an inference action which should be processed further on. In the case that an inference action uses a view its perfect Herbrand model is a super set of the complete domain layer. Therefore, it would not make sense to store this complete model in a store. Also in the case, that an inference action has several output stores, the class definitions of the stores can be used to choose different parts of the perfect model as their new contents.

For this selection, only those is-element-of literals, equality literals, and data literals are selected which fit to the class definitions of the store. For example, an *is-element-of literal* $e \in c$ in *H* is only chosen if the store contains a class definition for *c*. Similar, a data literal *e[...]* in *H* is only chosen if an is-element-of literal $e \varepsilon c'$ exists as element of *H* with c' is contained in the store. Figure 6 illustrates this. The input store contains a class definition for c_1 and the output store for c_2. For the sake of simplicity no attributes are defined. The input store contains one fact indicating that *a* is an element of c_1. The inference action is described by one clause which infers every *X* as an element of the class c_2 if it is an element of c_1. Therefore, the perfect Herbrand model of the inference action contains the two is-element-of literals $a \varepsilon c_1$ and $a \varepsilon c_2$ but only the second literal is chosen as new content of the output store.

Formally, the set of all is-element-of literals, equality literals, and data literals of the model of the elementary inference action are employed as the new contents of an output store *which are terms over the class definitions* of the store.

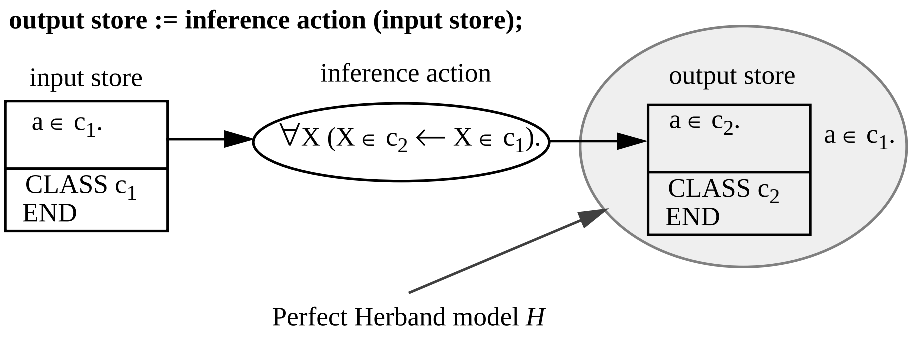

Fig. 6. An example for the semantics of an inference action.

6. KARL provides *modularisation* for the domain layer which is not discussed in the paper.
7. We do not discuss the case how a terminator is used to modify the domain layer because there is no technical difference.

Def. 4 Terms over a set of class definitions

Let *H* denote the perfect Herbrand model of a set of rules and definitions of an L-KARL alphabet. *CD* denotes a set of class definitions using the same L-KARL alphabet. *CD*(*H*) denotes the *set of all terms from H over CD*.

- An is-element-of term $e \in c$ of *H* is a term over the class definitions *CD*, if and only if CD contains a class definition for *c*.
- A data term $e[a_1 : T_1,..., a_r : T_r, s_1 :: \{ S_{11} ,..., S_{1m_1} \},..., s_t :: \{ S_{t1} ,..., S_{tm_t} \}]$ of *H*, *e* is an element-id-term, is a term over the class definitions *CD* if and only if *CD* contains a class definitions for a class *c* which defines all attributes $a_1,..., a_r, s_1,..., s_t$ for *c*. In addition, *H* must contain the element-of literal $e \in c$.
- A data term $c[a_1 : T_1,..., a_r : T_r, s_1 :: \{ S_{11} ,..., S_{1m_1} \},..., s_t :: \{ S_{t1} ,..., S_{tm_t} \}]$ of *H*, *c* is a class-id-term, is a term over the class definitions *CD*, if and only if *CD* contains a class definition for a class *c´* which defines all attributes $a_1,..., a_r, s_1,..., s_t$ for *c´*. In addition, *H* or *CD* must contain the is-a literal $c \leq c´$.
- Every equality term $o = o´$ of *H* is a term over the class definitions *CD*.

4.2 The Semantics of an Entire KARL Specification

As already mentioned an interpretation (D, $F^A = \{f_1^A,..., f_r^A\}$, $P^A = \{\varnothing_c, \varnothing_{c´},...\}$[8]) of a P-KARL program requires three ingredients: A universe *D* must be defined which provides the values for the variable assignments. A set F^A of functions must be given to interpret the function symbols used in elementary programs and a set P^A of relations must be provided to interpret the predicate symbols used in the formulae.

Def. 5 Universe of the P-KARL language

The *universe D* is defined as the *power set of the Herbrand base* of the union of the L-KARL languages used at the domain layer, inference layer, and for the domain view.

The Herbrand base contains all ground positive literals which can be expressed by these languages. Therefore, every perfect Herbrand model of an inference action is a subset of the Herbrand base and an element of its power set. A variable assignment to a P-KARL program assigns a set of ground literals to a variable. Such a variable and its assignment correspond to a store and its contents which is a subset of the current perfect Herbrand model of an inference action. Every variable of a P-KARL program corresponds to a store defined at the inference layer. The current contents of the stores are defined by the current values of the variables of the P-KARL program according to the given variable assignment. A *state* is therefore characterized by the current contents of the stores.

Every inference action defines a mapping between the current contents of its input and output stores. These mappings are used to interpret the *function symbols* in elementary P-KARL programs.

Def. 6 Interpretation of the function symbols of a P-KARL program

The function which is defined by an inference action *ia* is used to interpret the function symbol *ia* of P-KARL.

The three function symbols *Check*, *Create*, and *Prune* at the task layer in Figure 5 are interpreted by the three functions which are defined by the model-theoretic semantics of the inference actions *Check*, *Create*, and *Prune*.

Finally, the relations for interpreting the *predicate symbols* have to be defined. The formula $\varnothing_c(X)$ should be false for all states (i.e., variable assignments) for which the store *X* contains an is-element-of literal $e \in c$, where *e* is an arbitrary object denotation.

Def. 7 Interpretation of the predicate symbols of a P-KARL program

The unary predicate symbol $\varnothing_c$ is interpreted by the set of all elements of the universe D (which are sets) which do not contain an is-element-of literal $e \in c$, with *e* is an arbitrary object denotation.

Therefore, by defining a universe, functions, and relations the perfect Herbrand model semantics of L-KARL is used to define an interpretation for a P-KARL program.

5 The Operational Semantics and Its Influence on the Declarative Semantics

A *declarative semantics* of a language A defines the meaning of an expression in A by expressions of a second (mathematical) language B. These definitions need not to be constructive. For example, the minimal Herbrand model of a set of Horn clauses is defined as the intersection of all Herbrand models of this clause set which are infinitely many if function symbols are allowed. The purpose of the declarative semantics is to define a precise and detailed meaning of the language expressions. Because KARL should also support prototyping, an *operational semantics* based on its declarative semantics has been defined. The operational semantics of KARL

8. *c*, *c'*, etc. are the class names which occur in the class definitions of the stores.

allows the derivation of output facts based on a given set of input facts which supports the *validation* of a specification. The operational semantics for an inference action described by L-KARL is defined by a fixpoint operator [Llo87] which also has to consider the equalities which are defined by the equality-literals. The semantics of a P-KARL program is realized by a sequence operator which changes a state according to the specified control flow. For more details concerning the operational semantics and its implementation see [Ang93]. A significant restriction of the operational semantics is that it requires *finite* perfect Herbrand models. Otherwise the fixpoint operator does not terminate.

The declarative semantics of KARL has been developed with an eye on its operationalization. P-KARL programs are *regular* and *deterministic*. In addition, *recursion* is not available (i.e., the number of variables remain unchanged during execution). These restrictions significantly reduce the effort for operationalization. In the non-deterministic case all states must be regarded in parallel or the interpreter must choose a state and the operationalization would be incomplete. In the case of L-KARL, the language is restricted to a stratified set of clauses and only one specific model, the perfect Herbrand model, is regarded to define the truth value of formulae. The restriction on one specific model significantly reduces the effort for operationalization. Normally, three different sets of ground positive literals in regard to a given set of formula *S* exist. Literals which are entailed by *S*, literals which are entailed by *S* when they are negated, and literals which are neither a positive nor a negative consequence of *S*. In our case, the third set disappears and the truth value of a negated ground literal can be determined by checking whether the according positive literal is an element of the perfect Herbrand model. Work done in logic programming and deductive databases motivated our choice.

Whereas the restriction of P-KARL has not caused any problem in around twenty applications it was the restriction of L-KARL to Horn logic which often required artificial solutions. For example, when one wants to get the maximal element of a set it requires to introduce an artificial predicate (e.g., *not_max*) and double negation:

$$\forall X\, (X \in \text{max} \leftarrow X \in \text{set} \wedge \neg(X \in \textit{not\_max}).$$
$$\forall X\, \forall Y\, (X \in \textit{not\_max} \leftarrow X \in \text{set} \wedge Y \in \text{set} \wedge X < Y).$$

6 Tools Support and Applications

The tool set of MIKE provides support for three different types of activities: First, the *hypermedia tool MeMoKit* (Mediating Model Construction Kit, see [NeM93], [Neu93]) supports the semiformal specification of a knowledge-based system and bridges the gap to a formal specification. Starting with knowledge protocols achieved during knowledge elicitation sessions a semiformal description of the expertise can be built up. The main difference of this semiformal model of expertise to the KARL specification is that the atomar entities of the specification are defined by natural language. Therefore, the formal specification is achieved by supplementing these natural language descriptions with formal definitions. For this purpose, MeMoKit can be used as a graphical editor for KARL which additionally provides natural language descriptions and knowledge protocols as important documentations means to improve the understandability of the formal specification.

Second, an *interpreter* including debugging facilities for KARL is integrated in this graphical environment. A specification can be executed in this graphical environment, breakpoints can be defined, and the current contents of the stores can be inspected. The gist of the matter of this interpreter is an efficient bottom-up evaluation of L-KARL [Ang93]. Evaluation of specifications by testing always showed to be very helpful in validating them.

Third, work is in progress to support *design activities* (see [Lan94] and [LaS94]). This includes additional editors for constructing the design model, e.g., by refining parts of the model of expertise, as well as tools that allow to link design decisions to the requirements that motivate them. The operationalization of the design model is supported by mapping parts of the design model to code of a specific software target environment (C++), which also establishes the basis for evaluating the design model by running it as a hybrid prototype which consists of parts that are still executed using the KARL interpreter while others are already operational in the C++ target environment.

The development of different models of expertise for various expert system applications provided valuable feedback for developing the version of KARL which is described in this paper. Among others the following applications have been tackled: In [AFL+92a] and [AFL92c] two solutions for the Sisyphus-1 problem are described providing a complete model of expertise specification. For comparing different approaches for formally specifying the functionality of expert systems a simple scheduling task has been posed in [TrW93]. Here, a set of activities have to be scheduled to a set of time periods taking some restrictions into account. In [LFA93] a KARL solution for this problem is described resulting among others in the formalization of the problem solving method propose and exchange. In [KFG92] an approach for solving the problem of selecting appropriate scheduling algorithms from a library in a given project management context is discussed. As a third problem for comparing different knowledge level modelling approaches and languages an elevator configuration task has been posed [Yos92]. In [PFL+94] a KARL solution is described. This specification resulted in the formalization of the problem solving method propose and revise. The detailed specification of such a complex tasks gave us various insights in the advantages and disadvantages of some of the design decisions which we have made when developing KARL. In [Ang92] a complete specification of the problem solving method cover and differentiate has been provided based on the informal description which is given in [Mar88]. This problem solving method can be used to solve problems in which a solution has to be identified from a given set of solutions, like e.g. in diagnosis tasks. The Board-game method is a refinement of chronological backtracking for problems, in which a fixed set of pieces can be moved between locations in order to reach a goal state [EST+92]. Using the informal description given in [EST+92] as well as the available CLIPS code the board-game method has been formally specified [FEM+93]. This case study is a typical example how KARL can be used for re-engineering already implemented problem solving methods.

7 Comparison

In the following, we compare briefly KARL with specification languages from knowledge engineering ($(ML)^2$ and DESIRE), information system development (Telos and TROLL), and software engineering (VDM and Z). A comparison of KARL with most existing knowledge specification languages can be found in [FeH94] and [AFL92b] contains a comparison of KARL with INCOME, a language for specifying information systems.

The main difference between KARL and $(ML)^2$ [HaB92] grounds in the fact that $(ML)^2$ is a language which aims only at formalizing models of expertise of kbs and not at operationalization. It uses full first-order logic to specify static knowledge and regards all possible interpretations to determine the truth value of formulae. A possible operationalization of $(ML)^2$ would require theorem-proving techniques. This operationalization would not be complete and less efficient than an operationalization of Horn logic as used by KARL. Otherwise as already mentioned, the restriction to Horn logic as done by KARL often requires unnatural specifications and hinder the transition from an informal to a formal specification. A further difference lies in the fact that KARL uses an object-oriented customization of first-order logic to express domain and inference knowledge whereas $(ML)^2$ provides pure predicate logic for this purpose. Therefore, epistemological different types of knowledge like concepts, attributes, domain and range restrictions are uniformly represented by predicates in $(ML)^2$. Both languages use dynamic logic for specifying procedural knowledge. Otherwise, there is a significant difference in how both languages use dynamic logic. In KARL, an inference action is represented by a function. In $(ML)^2$, every inference actions is represented by a predicate and the logical description of them is integrated into dynamic logic by clauses having this predicate as a literal. As a consequence, $(ML)^2$ requires *quantified* dynamic logic because the logical description of inferences require quantifiers whereas for P-KARL a less expressive variant of dynamic logic is sufficient because every formula contains free variables only (cf. [Koz90]). *DESIRE* [LPT93] is a further knowledge specification languages but differs significantly from KARL in the conceptual model they apply as DESIRE does not rely on the KADS model of expertise. The main difference is that DESIRE uses meta-level architecture for dynamic control based on *temporal logic*. Therefore, the control flow is not procedurally described but a set of meta-level formulae can constrain possible control flows. Semantically, DESIRE represents its partial reasoning by a temporal logic with a fixed discrete set of time points where the control flow is represented as a trace of different points in time. DESIRE provides no means to reason about composition of actions like sequences or loops.

Telos [MBJ+93] which has envolved from RML (Requirements Modeling Language, see [GMB94]) is a language to support the development of information systems. It views the specification as an object-oriented knowledge base supplemented by constraints and rules. An intervall-oriented temporal formalism is provided to express temporary features of these objects and their relationships. Telos provides the operations RETRIEVE and ASK for querying this knowledge base. Besides some technical details, L-KARL and the analogous part of Telos are very close in spirit. As in KARL attributes and classes are treated as first-order citizens but KARL provides a more explicit notion of complex objects. The main difference between both languages concerns the representation of dynamics. KARL has no explicit means to represent temporal knowledge and Telos includes no representation of procedural knowledge as the system is viewed as a static knowledge base and not as a program.

TROLL [Jun93] is a language for formally specifying information systems and is used during requirements specification or conceptual modelling—a phase in the development of information systems which directly corresponds to knowledge acquisition. In TROLL an information system is modelled as a community of interacting objects where each object covers structural as well as behavioural aspects. A basic characteristic of TROLL is the use of temporal logic for specifying e.g. the admissible state sequences of objects. In [Jun93] the formal semantics of TROLL is given using the so-called Object Specification Logic [SSC92]. However, a complete operationalization of TROLL is currently not available. Again a major difference between KARL and TROLL is the fact that the latter aim primarily at the specification of objects and their behaviour. The behaviour is characterized by rather simple operations which change the state of the corresponding objects and may cause the state change of other objects. However, the specification of complex application functions, which are described in

KARL on the task and inference layer, is not addressed in these languages. On the other hand, it should be clear that e.g. notions like dynamic integrity constraints as offered by TROLL could be used to enhance the specification of the domain layer of a KARL model of expertise. The main difference between KARL and TROLL is again the representation of dynamics. Whereas the later constrains possible control flows by use of temporal logic, KARL uses dynamic logic to explicitly define the control flow in a procedural manner. As already mentioned in section 1.3, this design decision in KARL was taken as experiences in large expert system projects have shown that great problems arise if the control flow is implicitly specified.

VDM and Z are formal specification languages developed by the software engineering community. Both languages can report several applications in the development of large software products. *VDM* [Jon90] describes a system by a data model 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. Pre-conditions, post-conditions, and invariants are means to specify 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. *Z* is based on typed set theory. Static and dynamic aspects of a system are uniformly described by so-called schemes. Large specifications can be built up by combining several of these schemes. A schema describes a data type or an operation by means of pre-conditions, post-conditions, and invariants. For this purpose, sets, relations, and functions can be defined using a predicate logic like language. Its mathematical semantics is defined in [Spi88] and [Spi92] defines a standard for Z. Comparing KARL with VDM and Z, three main differences arise:

- Neither VDM nor Z aim on prototyping by executable specifications. Our experience with KARL showed the usefulness of executing specifications in two dimensions. First, it helps to understand what is specified. That is, it helps to detect "programming errors" in a specification. Second, it helps to check whether the specification meets the desired functionality, that is, whether the semantics of the formal specification meets the informally defined requirements.
- VDM and Z define a rigorous syntactic and (several years later) semantical framework to mathematically describe systems 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 partial function which is correct but a very low level of describing a system.
- Recently, Z and VDM [BFL+94] provide proof theories which allows to formally prove properties of specifications. This work remains still to be done for KARL.

Several semiformal specification techniques have been developed in software engineering. In [FAL93] it is described how KARL can be used to formalize and operationalize software specifications using structured analysis techniques like data flow diagrams [You89]. This shows that the application of KARL is not restricted to the development of KBS, instead application systems being developed with a classical software engineering technique like structured analysis can be formally specified in KARL as well. Otherwise, there are no means to specify real-time problems in KARL.

Conclusion

The paper presented the knowledge specification language KARL and its declarative semantics. Meanwhile KARL has been successfully applied in around twenty academic case studies for specifying knowledge-based systems (kbs). Its semantics enables the precise and detailed specification of a kbs without referring to implementational details which are not of interest during the knowledge acquisition process. Therefore, it defines an intermediate level between informal and semiformal specification at the one hand and the efficient implementation of a kbs at the other hand. The KARL model of expertise contains the description of domain knowledge, inference knowledge, and task knowledge (i.e., procedural control knowledge) and allow a smooth transition from semiformal to formal specifications.

KARL integrates two different types of logic: The sublanguage L-KARL, which is based on object-oriented logics, combines frames and logic to define terminological as well as assertional knowledge. The sublanguage P-KARL, which is a variant of dynamic logic, is used to express knowledge about the control flow of a problem-solving method in a procedural manner. The validation of KARL specifications is enabled by the definition of an operational semantics which coincides with the declarative semantics and by the implementation of a KARL debugger. The development of a *proof theory* for KARL by defining a set of complete and correct derivation rules is an important topic for future research.

Acknowledgement

We thank Manfred Aben, Frank van Harmelen, Dieter Landes, Susanne Neubert, and Jan Treur for fruitful discussions and helpful comments on early drafts of the paper.

Bibliography

[AFL+92a] J. Angele, D. Fensel, D. Landes, and R. Studer: An Assignment Problem in Sisyphus - No Problem with KARL. In M. Linster (ed.): Sisyphus ´91: Models of Problem Solving, Arbeitspapiere der GMD, no 630, March 1992..

[AFL92b] J. Angele, D. Fensel, and D. Landes: Two Languages to Do the Same? In *Proceedings of the 2nd Workshop Informationssysteme und Künstliche Intelligenz*, February 24-26, 1992, Ulm, R. Studer (ed.), Informatik-Fachberichte, no 303, Springer-Verlag, Berlin, 1992.

[AFL92c] J. Angele, D. Fensel und D. Landes: An Executable Model at the Knowledge Level for the Office-Assignment Task. In [Lin92].

[AFL+93] 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.

[AFS94] J. Angele, D. Fensel, and R. Studer: The Model of Expertise in KARL. In *Proceedings of the 2nd World Congress on Expert Systems*, Lisbon/Estoril, Portugal, January 10-14, 1994.

[Ang92] J. Angele: Cover and Differentiate Remodeled in KARL. In *Proceedings of the 2nd KADS User Meeting*, Munich, February 17-18, 1992, C. Bauer et al. (eds.), Interpretation Models for KADS - Proceedings of the 2nd KADS User Meeting (KUM´92), GMD report, no 212, 1992.

[Ang93] J. Angele: Operationalisierung des Modells der Expertise mit KARL (Operationalization of a Model of Expertise with KARL), Infix, St. Augustin, 1993 (in German).

[ARS92] S. Aitken, H. Reichgelt, and N. Shadbolt: Representing KADS Models in QIL, AI Group, University of Nottingham, Working Paper WP-006, 1992.

[BaS89] J. Bachant and F. Soloway: The Engineering of XCON, *Communications of the ACM*, vol 32, no 3, March 1989.

[BFL+94] J. C. Bicarregui, J. S. Fitzgerald, P. A. Lindsay, R. Moore, and B. Ritchie: *Proof in VDM: A Practitioner´s Guide*, Springer Verlag, Berlin, 1994.

[EST+92] H. Eriksson, Y. Shahar, S. W. Tu, A. R. Puerta, and M. A. Musen: Task modeling with reusable problem-solving methods. In *Proceedings of the 7th Banff Knowledge Acquisition for Knowledge-Based Systems Workshop*, Banff, Canada, October 11–16, 1992.

[FeH94] D. Fensel and F. van Harmelen: A Comparison of Languages which Operationalize and Formalize KADS Models of Expertise. *The Knowledge Engineering Review*, vol 9, no 2, June 1994.

[FEM+93] D. Fensel, H. Eriksson, M. A. Musen, and R. Studer: Description and Formalization of Problem-Solving Methods for Reusability: A Case Study. In *Complement Proceedings of the 7th European Knowledge Acquisition Workshop (EKAW´93)*, Toulouse, France, September 6-10, 1993.

[Fen93] D. Fensel: *The Knowledge Acquisition and Representation Language KARL*, Ph.D. thesis, University of Karlsruhe, 1993. To appear Kluwer Academic Publisher, Boston, 1995.

[GMB94] S. Greenspan, J. Mylopoulus, and A. Borgida: On Formal Requirements Modeling Languages: RML Revisited. In *Proceedings of the 16th International Conference on Software Engineering (ICSE´94)*, Sorrento, Italy, May 16-21, 1994.

[HaB92] F. v. Harmelen and J. Balder: $(ML)^2$: A Formal Language for KADS Conceptual Models. In *Knowledge Acquisition*, vol 4, no 1, 1992.

[Har84] D. Harel: Dynamic Logic. In D. Gabby et al. (eds.), *Handook of Philosophical Logic, vol. II, Extensions of Classical Logic*, Publishing Company, Dordrecht (NL), 1984, pp. 497-604.

[Jon90] C.B. Jones: *Systematic Software Development Using VDM*, 2nd ed., Prentice Hall, 1990.

[JoS92] W. Jonker and J.W. Spee: Yet Another Formalisation of KADS Conceptual Models. In *Proceedings of the 6th European Knowledge Acquisition for Knowledge-Based Systems Workshop (EKAW-92)*, May 18-22, Heidelberg/ Kaiserslautern, T. Wetter et al. (eds.), *Current Developments in Knowledge Acquisition*, Lecture Notes in Artificial Intelligence, no 599, Springer-Verlag, Berlin, 1992.

[Jun93] R. Jungclaus: *Modeling of Dynamic Object Systems - A Logic Based Approach*, Vieweg Verlag, Braunschweig, 1993.

[KaV93] W. Karbach and A. Voß: MODEL-K For Prototyping and Strategic Reasoning at the Knowledge Level. In J.-M. David, J.-P. Krivine, and R. Simmons (eds.), *Second Generation Expert Systems*, Springer-Verlag, Berlin, 1993.

[KFG92] R. Köppen, D. Fensel, and J. Geidel: Modelling the Selection of Scheduling Algorithms with KARL. In *Proceedings of the 2nd KADS User Meeting*, Munich, February 17-18, 1992, C. Bauer et al. (eds.), Interpretation Models for KADS - Proceedings of the 2nd KADS User Meeting (KUM´92), GMD report, no 212, 1992.

[KLW93] M. Kifer, G. Lausen, and J. Wu: Logical Foundations of Object-Oriented and Frame-Based Languages, technical report 93/06, Department of Computer Science, SUNY at Stony Brook, NY, April 1993. To appear in *Journal of the ACM*.

[Koz90] D. Kozen: Logics of Programs. In J. v. Leeuwen (ed.), *Handbook of Theoretical Computer Science*, Elsevier Science Publ., B. V., Amsterdam, 1990.

[Lan94] D. Landes: DesignKARL - A Language for the Design of Knowledge-Based Systems. In *Proceedings of the 6th International Conference on Software Engineering and Knowledge Engineering SEKE'94*, Jurmala, Latvia, June 20-23, 1994.

[LaS94] D. Landes and R. Studer: The Design Process in MIKE. In *Proceedings of the 8th Knowledge Acquisition for Knowledge-Based Systems Workshop KAW´94*, Banff, Canada, January 30 - February 5, 1994.

[LFA93] D. Landes, D. Fensel, and J. Angele: Formalizing and Operationalizing a Design Task with KARL. In [TrW93].

[Lin92] M. Linster (ed.): Sisyphus ´92: Models of Problem Solving, Arbeitspapiere der GMD, no 663, July 1992.

[Lin93] M. Linster: Using OMOS to Represent KADS Conceptual Models. In [SWB93].

[Llo87] J.W. Lloyd: *Foundations of Logic Programming, 2nd Editon*, Springer-Verlag, Berlin, 1987.

[LPT93] Izak van Langevelde, A. Philipsen, and J. Treur: A Compositional Architecture for Simple Design Formally Specified in DESIRE. In [TrW93].

[Mar88] S. Marcus (ed.): *Automating Knowledge Acquisition for Experts Systems*, Kluwer Academic Publisher, Boston, 1988.

[MBJ+93] J. Mylopoulos, A. Borgida, M. Jarke, and M. Koubarakis: Representing Knowledge About Information Systems in Telos. In M. Jarke (ed.), *Database Application Engineering with DAIDA*, research reports ESPRIT, project 892, DAIDA, vol 1, Springer-Verlag, Berlin, 1993.

[NeM93] S. Neubert and F. Maurer: A Tool for Model Based Knowledge Engineering. *In Proceedings of the 13th International Conference AI, Expert Systems, Natural Language (Avignon´93)*, May 24-28, Avignon, 1993.

[Neu93] S. Neubert: Model Construction in MIKE (Model-Based and Incremental Knowledge Engineering). In N. Aussenac et al. (eds.), *Knowledge Acquisition for Knowledge-Based Systems, Proceedings of the 7th European Workshop (EKAW´93,* Toulouse, France, September 6-10, 1993), Lecture Notes in AI no 723, Springer-Verlag, Berlin, 1993.

[New82] A. Newell: The Knowledge Level, *Artificial Intelligence*, vol 18, 1982.

[PET+92] A. R. Puerta, J. W. Egar, S. W. Tu, and M. A. Musen: A Multiple-Method Knowledge-Acquisition Shell For The Automatic Generation Of Knowledge-Acquisition Tools. In *Knowledge Acquisition*, vol 4, no 2, 1992.

[PFL+94] K. Poeck, D. Fensel, D. Landes, and J. Angele: Combining KARL and Configurable Role Limiting Methods for Configuring Elevator Systems. In *Proceedings of the 8th Banff Knowledge Acquisition for Knowledge-Based System Workshop (KAW´94)*, Banff, Canada, Januar 30th - February 4th, 1994.

[Prz88] 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.

[Pup93] F. Puppe: *Systematic Introduction to Expert Systems: Knowledge Representation and Problem-Solving Methods*, Springer-Verlag, Berlin, 1993.

[SBJ87] E. Soloway, J. Bachant, and K. Jensen: Assessing the Maintainability of XCON-in-RIME: Coping with the Problems of a VERY Large Rule-Base. In *Proceedings of 6th National Conference on AI* (AAAI-87), Seattle, Washington, July 13-17, 1987, pp. 824-829.

[SpA91] V. Sperschneider and G. Antoniou: *Logic: A Foundation for Computer Science*, Addison-Wesley Pub., Wokingham, England, 1991.

[Spi88] J.M. Spivey: *Understanding Z. A Specification Language and Its Formal Semantics*, Cambridge University Press, Cambridge, 1988.

[Spi92] J.M. Spivey: *The Z Notation. A Reference Manual*, 2nd ed., Prentice Hall, New York 1992.

[SSC92] A. Sernadas, C. Sernadas, and J.F. Costa: *Object Specification Logic*. Research Report INESC/DMIST, University of Lisbon, 1992. To appear in *Journal of Logic and Computation*.

[SWB93] G. Schreiber, B. Wielinga, and J. Breuker (eds.): *KADS. A Principled Approach to Knowledge-Based System Development*, Knowledge-Based Systems, vol 11, Academic Press, London, 1993.

[TrW93] J. Treur and Th. Wetter (eds.): *Formal Specification of Complex Reasoning Systems*, Ellis Horwood, New York, 1993.

[Ull88] J. D. Ullman: *Principles of Database and Knowledge-Base Systems, vol I*, Computer Sciences Press, Rockville, Maryland, 1988.

[VoV93] H. Voss and A. Voss: Reuse-Oriented Knowledge Engineering with MoMo. In *Proceedings of the 5th International Conference on Software Engineering and Knowledge Engineering (SEKE´93)*, San Fransisco Bay, June 14-18, 1993.

[Wet90] T. Wetter: First Order Logic Foundation of the KADS Conceptual Model. In B. Wielinga et al. (eds.), *Current Trends in Knowledge Acquisition*, IOS Press, Amsterdam, 1990.

[Yos92] G.R. Yost: *Configuring Elevator Systems.* Technical report, Digital Equipment Co., Marlboro, Massachusetts, 1992.

[You89] E. Yourdon: *Modern Structured Analysis*, Prentice-Hall, Englewood Cliffs, 1989.