to appear in IEEE Transactions in Knowledge and Data Engineering (TKDE)

The Knowledge Acquisition and Representation Language KARL

Dieter Fensel*, Jürgen Angele+, Rudi Studer*

*Institut AIFB, Universität Karlsruhe, D-76128 Karlsruhe,
e-mail: {fensel, studer}@aifb.uni-karlsruhe.de
+Institut für Angewandte Informatik, Fachhochschule Braunschweig, D-38302 Wolfenbüttel,
e-mail: angele@informatik.fh-wolfenbüttel.de

Abstract

The *Knowledge Acquisition and Representation Language* (KARL) combines a description of a knowledge-based system at the conceptual level (a so-called *model of expertise*) with a description at a formal and executable level. Thus, KARL allows the precise and unique specification of the functionality of a knowledge-based system independent of any implementational details. A KARL model of expertise contains the description of domain knowledge, inference knowledge, and procedural control knowledge. For capturing these different types of knowledge KARL provides corresponding modeling primitives that are based on Frame-logic and Dynamic Logic. A declarative semantics for a complete KARL model of expertise is given by a novel combination of these two types of logic. In addition, an operational definition of this semantics, which relies on a fixpoint approach, is given. This operational semantics defines the basis for the implementation of the KARL interpreter which includes appropriate algorithms for efficiently evaluating KARL specifications. This enables the evaluation of KARL specifications by means of testing.

1 Introduction

During the last years a lot of informal, semiformal, and formal description techniques have been developed for specifying the functionality of a KBS in an implementation independent way. Such techniques fulfil the same purpose for KBS-development as dataflow diagrams, entity-relationship diagrams, state-transition diagrams, etc. in software engineering and information systems engineering.

Informal techniques enable the specification of a system at a conceptual level in an easy and understandable manner. On the other hand, it is familiar 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 [LPT93a], FORKADS [Wet90], $K_{BS}SF$ [JoS92], $(ML)^2$ [HaB92], MODEL-K [KaV93], MODEL/ KADS [Bar93], MoMo [VoV93], OMOS [Lin93], QIL [ARS92], and KARL have been developed to improve the result of the specification phase by supplementing informal descriptions. A description of most of these languages and their comparison can be found in [FeH94]. A formal description reduces the vagueness and ambiguity of natural-language descriptions by enforcing preciseness without dealing with implementational decisions. Such a formalized description allows formal proofs of properties of the specified knowledge. Furthermore, some of these languages provide an automatic mapping to an (inefficient) operational description to permit testing as a means for knowledge evaluation. The main concern of the *Knowledge Acquisition and Representation Language (KARL)* is to provide integrated means for a specificying knowledge at a conceptual, formal and operational level.

First, KARL provides appropriate modelling primitives to allow knowledge specifications at the *knowledge level*. It distinguishes several types of knowledge and defines different language primitives for them. The conceptual model underlying a specification in KARL is a modification of the KADS *model of expertise* [SWB93]. Therefore, a smooth transition from semiformal to formal specifications is possible. Both use the same conceptual framework and the same modelling primitives. The difference lies in the fact, that these modelling primitives become a defined semantics when applying KARL to formally specify them. The formal description in KARL is reached by a refinement step which complements informal descriptions in natural languages with formal definitions. The graphical representation of most modelling primitives support this transition and the usefulness of KARL as a means for communication with users and experts. In addition, they allow the implementation of a graphical editor for KARL.

Secondly, KARL is a *formal* knowledge specification language. That is, it has a declarative semantics. The advantages of a formal and declarative semantics are: The language primitives get a defined meaning which allows a precise and unequivocal description of knowledge. A declarative semantics should be easier to understand than its operationalization as the later one deals with two aspects: *what* is the meaning of an expression and *how* can it be computed. The declarative semantics can be used as a base line for its operationalization. In fact, the declarative semantics of KARL was used as a specification for the interpreter which was developed for KARL. The semantics must include the representation of static and dynamic (i.e., procedural) knowledge. In fact, the gist of the matter of the semantics of KARL was the integration of static and dynamic knowledge.

Thirdly, KARL is an *operational* knowledge specification language. That is, its semantics has been operationalized and a debugger has been implemented. This operational description can support knowledge evaluation via testing and debugging. As significant parts of the expertise of a human expert is hidden in his skills and exists as implicit knowledge, only. The elicitation and acquisition of this knowledge as well as its formal description requires a modelling activity. As every model building process, knowledge acquisition is incremental, in principle infinite, and faulty. The usefulness of testing (i.e., operational specifications) for evaluating specifications is well-known in the domain of software engineering (cf. [Flo84]). Therefore, we defined KARL in a way that its mechanization became possible. A debugger allows the stepwise evaluation of knowledge specifications.

The development of the language KARL is part of the *MIKE-approach* (*Model-based and Incremental Knowledge Engineering*) [AFL+93]. The overall goal of the MIKE project is the definition of a development method for knowledge-based systems (KBS) covering all steps from initial knowledge acquisition to design and implementation. MIKE provides a cascade of models to bridge the gap between informal requirements and expertise at the one hand and an implemented knolwedge-based system at the other hand. The first model is the so-called *elicitation model* which provides a natural language description of the expertise having been collected from the expert or e.g. from text books. Using the elicitation model as a basis the so-called *structure model* [Neu93] is built-up during the interpretation phase which succeeds the elicitation phase in the MIKE process model. The structure model provides a semiformal representation of the expertise by organizing the expertise in a hypertext network which offers predefined types of nodes and arcs. By using the structure model as a basis the KARL model of expertise is built up during the formalization phase. Thus the task and domain knowledge being semiformally described in the structure model is represented in a unambiguous and formal way. Since KARL is also an executable language MIKE offers a formal *and* operational model of expertise as the result of the knowledge acquisition phase. The KARL model of expertise does not take into account non-functional requirements like e.g. maintainability. This type of requirements is considered in the design phase of MIKE resulting in the so-called *design model* [Lan94]. For specifying the design model KARL is extended to DesignKARL which offers additional language primitives like data structures and algorithms. This design model is then the basis for implementing the final KBS. A special characteristic of the MIKE approach is the fact that all the different models are related to each other in a well-defined way thus providing means for e.g. tracing back parts of the design model to the formal KARL model of expertise or the semiformal structure model.

The contents of the paper are organized as follows. The modelling primitives of the language KARL are introduced in section two. Then the declarative semantics of the sublanguages L-KARL and P-KARL and finally the semantics of the complete language KARL are given in sections three. The paper does not only describe the semantics but give the main reasons for our choices. The declarative semantics of KARL has been developed with an eye to its operationalization. Section four discusses this mechanization of KARL. Then, the tool environment of KARL and some of its applications are sketched in section five. Finally, some applications of the language are discussed in section six and the last section provides a comparison of KARL with some other specification languages.

2 The Modelling Primitives of 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 domain and inference knowledge. The sublanguage Procedural-KARL (P-KARL) is used to specify the control flow of the problem-solving process. 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 knowledge engineering [Lin92]. An assignment problem was posed, in which employees are assigned to office places with several requirements to be met. Extracts of this example are used to illustrate KARL.

2.1 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 for developing expert system—at least in Europe—we tried to keep the KARL model as close as possible to the KADS model of expertise. On the other hand, this model is only defined informally and refinement and modifications are natural results of every formalization process. For the sake of self containment we briefly sketch the KADS model of expertise before we show how this model was realized 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. It distinguishes static knowledge and three types of control knowledge. Because there is still significant disagreement about the third type of control knowledge (i.e., the *strategic layer*) we have neither considered it for the KARL model nor will we discuss it further in this paper. However, we followed KADS in distinguishing domain, inference, and task layers.

The *domain layer* 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 in which 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.

The *inference 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. It 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 it defines. 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. In general, knowledge roles have to be connected with domain classes and inference actions have to be connected with the knowledge required for such an inference.

The *task layer* 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 separation of domain knowledge and the different types of control knowledge should enable two kinds of reuse. The domain knowledge could be reused for different tasks and the domain-independent control knowledge at inference and task layer could be reused for similar tasks in different domains (for example, fault diagnosis of mechanic devices and medical diagnosis for humans). The domain-independent control knowledge of a knowledge-based system at the inference and task layer is called an interpretation model or a *problem-solving method*. It defines in generic terms a behavioral model of the systems` problem-solving capability. Libraries of such reusable problem-solving methods are provided by [Ben95] and [BvV94].

The Refined Model of Expertise of KARL

Originally, KADS proposed KL-ONE as a language for the domain layer. KL-ONE defines a very restricted set of language primitives which enables strong characterizations of decidability and efficiency of reasoning. Yet, for a specification language, a broad syntactical variety of modelling primitives appears 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) [KLW95]. Terminological knowledge can be described by a taxonomy of classes. Attributes can be defined for each class and are inherited according to the taxonomy. Additional knowledge can be described with logical formulae. A domain layer is structured and hierarchically ordered by the is-a hierarchy between classes and by a module hierarchy.

In addition to its use at the domain layer, L-KARL is used to specify the logical relationship defined by an inference action at the inference layer. Extending KADS, the conceptual modelling primitives of 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 *task-specific* and *problem-solving-method-specific* terminologies independent of 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]. Large specifications are therefore manageable in KARL.

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. L-KARL distinguish three types of knolwedge roles. A *view* could be used to provide domain knowledge for an inference action at the inference layer. Complementarily, 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. Finally, a *store* is used to model the data and knowledge flow between inferences. It has no connection to the domain layer but is used to connect two or more inference actions.

Procedural knowledge: The sublanguage *Procedural-KARL (P-KARL)* is used for specifying 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 only described informally. The syntax of P-KARL is just sugar on top of *dynamic logic* (cf. [Har84], [Koz90]) which defines the essence of the language. We chose dynamic logic because it is a well-known means for declaratively describing procedural programs.

The Sisyphus Example

During the following, we provide an informal and graphical representation of our running example. In the following section, the elementary elements of the models will be defined by the logical language. 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 process 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 who are supervisors (named *Bosses*). Each class is further described by attributes. Employees are described by their names, whether they smoke, and by the set of other employees with whom they can share a room (cf. the attribute *fit-together*). The places are described by their room numbers. The attribute *placement* should represent the solution, that is, for each employee his or her work place. The class *Bosses* inherit the attributes of the class *Employees* via the is-a link.

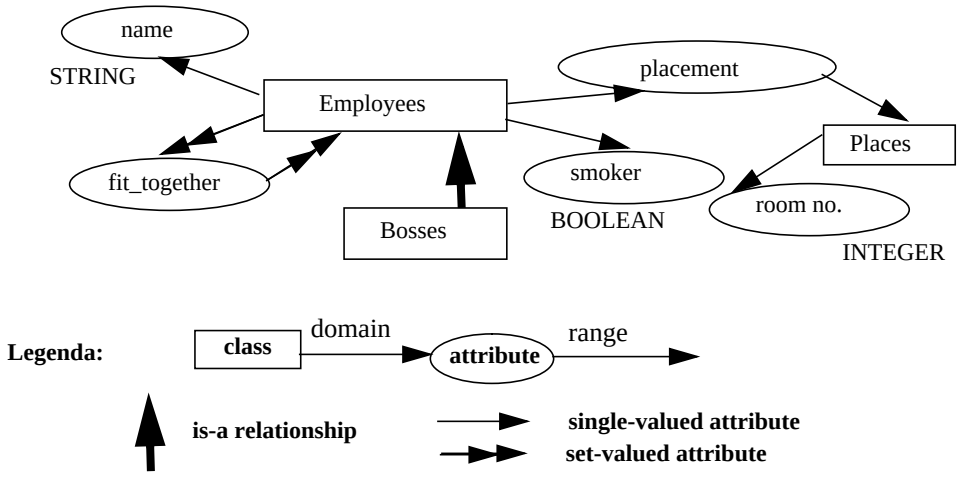

Fig. 1 The domain layer of the Sisyphus example.

The inference layer as shown in Figure 2 contains the elementary inference steps and knowledge roles of the problem-solving method. The inference actions *create* takes *components*, *slots,* and *assignments* as input and delivers new (extended) assingment. *Components* and *Slots* are domain views that must be mapped on domain knowledge. In our example, *Employees* will be mapped on *Components* and *Places* on *Slots*. The assignments are handled by the store *Assignments*. *Prune* eliminates incorrect assignments by using the

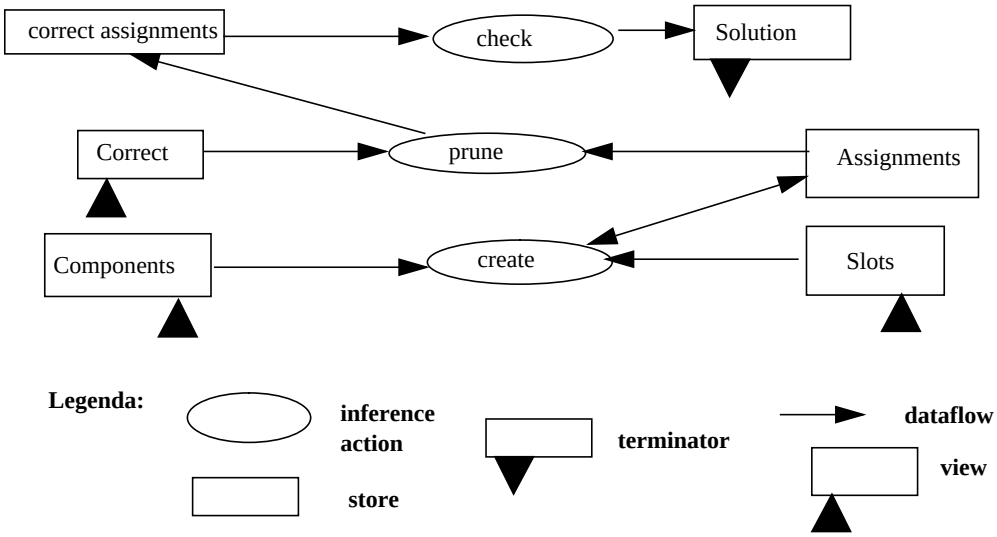

Fig. 2 The inference layer of the Sisyphus example.

domain knowledge delivered by the domain view *Correct*. *Check* searches for valid solutions which are saved via the terminator *Solution* at the domain layer.

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 (i.e., when $\varnothing_{solution}$(Solution) is no longer evaluated to be true). This simple control flow uns in an infinite loop if no solution exists, but we want to keep the example as simple as possible.

2.2 The Domain Layer: Logical-KARL (L-KARL)

L-KARL is a customization of Frame-logic (F-logic) [KLW95]. F-logic and L-KARL enrich the modelling primitives of first-order logic through syntactic modifications, but preserve its model-theoretical semantics. In this way, ideas of semantical and object-oriented data models are integrated into a logical framework which enables the declarative description of terminological and assertional knowledge.

L-KARL distinguishes classes, elements, and values. *Classes* refer to sets of real-world

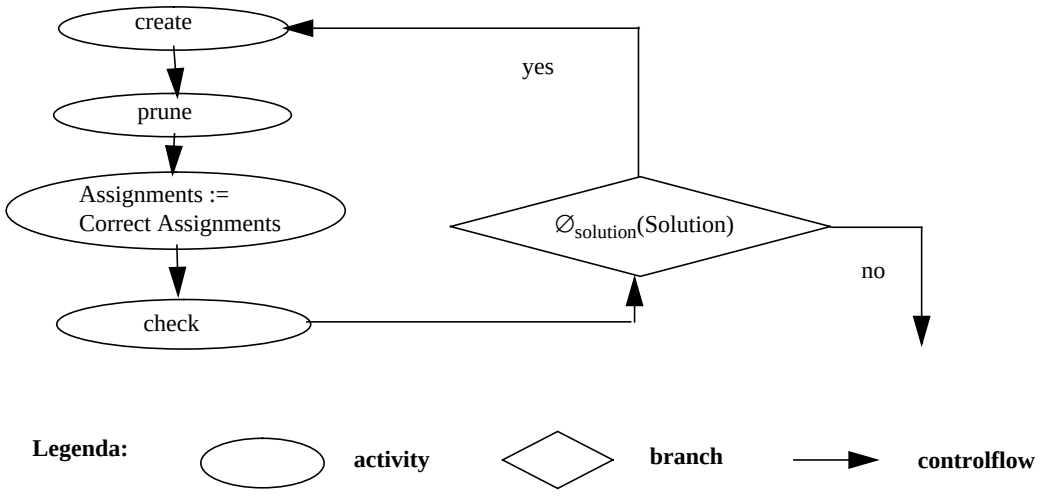

Fig. 3 The task layer of the Sisyphus example.

objects with common features. Classes and elements can be described by attribute values. Attribute and class definitions, together with an is-a hierarchy and multiple attribute inheritance can be applied to describe *terminological knowledge*. These attributes have defined domains and ranges and can be applied to describe elements of classes or to describe the classes themselves. *Elements* refer to real-world objects whereas *values* are only used to describe such objects.[1] *Intentional* and *factual knowledge* are described through 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 of generating new objects is based on O- and F-Logic (cf. [KiW89], [KLW95]). 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 ranges. Classes are arranged in an is-a hierarchy with multiple attribute inheritance. Attributes can be single-valued or set-valued. As mentioned above, attributes can be used to describe elements as well as classes. They have defined domain types and range types. In general, 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. Therefore, the specification of attributes via class definitions 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. That is, a minimal Herbrand model is only regarded as valid semantics when it fulfils these restrictions (see section two).

Figure 4 provides the definition of the terminology of the domain layer in our example. The three classes together with their attributes are defined in L-KARL.

The literals of logical expressions in L-KARL are *is-element-of literals* which state that objects are elements of classes; *is-a literals* which describe sub- and superset relationships between the classes; *equality literals* which denote 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 for describing relationships between classes, objects, and values is Horn logic with equality and function symbols extended by stratified negation (cf. [Prz88], [Ull88]).

1. See [Kim90] and [Bee90] for an introduction in object orientation and its integration with logic.

```
/* The class employees models the employees */
CLASS     employees
          ELEMENT_ATT
                    name  :          {STRING};
                    fit_together ::  {employees};
                    placement :      {places};
                    smoker :         {BOOLEAN}
END;
/* The class places models the places in the rooms. */
CLASS     places
          ELEMENT_ATT
                    room_no :        {INTEGER};
END;
/* The class bosses models the heads of  projects. */
CLASS     bosses
          ISA       employees;
END;
```

Fig. 4 The terminological knowledge of the Sisyphus example defined by using KARL.

Def. 1 (Positive Literal in L-KARL)
A positive literal in L-KARL is either:

- an *is-element-of literal* $e \in c$ 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-literals. 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. Furthermore, the arguments of a predicate are named and typed.

An example for a logical formulae is provided by the following clause that expresses that two employees fit together if nobody of them is a boss (i.e., bosses get single rooms) and if both smoke or neither smokes:

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

In addition, 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 refer to case-specific input of the user, describing his specific problem. Output attributes store the results of a problem-solving process. For reasons of limited space, we will not discuss this distinction further (see [Fen95b] 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. Table 1 and 2 provide a definition of the factual domain knowledge of our example.

Table 1. 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

Table 2. class: bosses

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

The rows in the tables correspond to ground clauses like:

fvh [*name* : *"Frank van Harmelen"*, *smoker*: NO] $\wedge$ *fvh* $\in$ *employees*

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

fvh [*name* : *"Frank van Harmelen"*, *smoker*: NO] $\in$ *employees.*

2.3 The Inference Layer and the Mapping between Domain and Inference Layers: Logical-KARL (L-KARL)

L-KARL is also used to specify the logical inferences at the inference layer and their connection with the domain layer. We will show this in part. In fact, we give the definition of the inference action *prune,* its input store *Assignments,* its output store *Correct assignments,* and its view *Correct* in Figure 5. *Prune* deletes all assignments which describe incorrect assignments. The inference action is very easy to specify as shown in Figure 5. Every *Assignments* that is not an incorrect assignments is a correct assignment. That is, such a state is re-examined later in the problem-solving process until all components have been assigned to a slot. Domain knowledge is required by the inference action in order to decide whether a state is correct or not. The required domain knowledge is delivered by the view *Correct*. It specifies when a state is an incorrect state. An assignment *Z* having two assignments A_1 and A_2 where different employees are assigned to the same room is a assignments state if the assigned employees do not fit together. The set-valued attribute *fit_together* was defined at the domain layer.

2.4 The Task Layer: Procedural-KARL (P-KARL)

Experience with XCON [SBJ87] showed that great problems arise if the control flow is only specified implicitly. 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 subactivities and that this knowledge is required to solve a task efficiently. Therefore, this knowledge was implicitly encoded in the rule formalism. Similar experiences have been made with PROLOG where the order of clauses and literals and primitives like the cut are used to implicitly specify the control flow. This need for specification of the control flow leads to the development of the sublanguage P-KARL. The control flow is specified in a way similar to that of procedural programming languages.

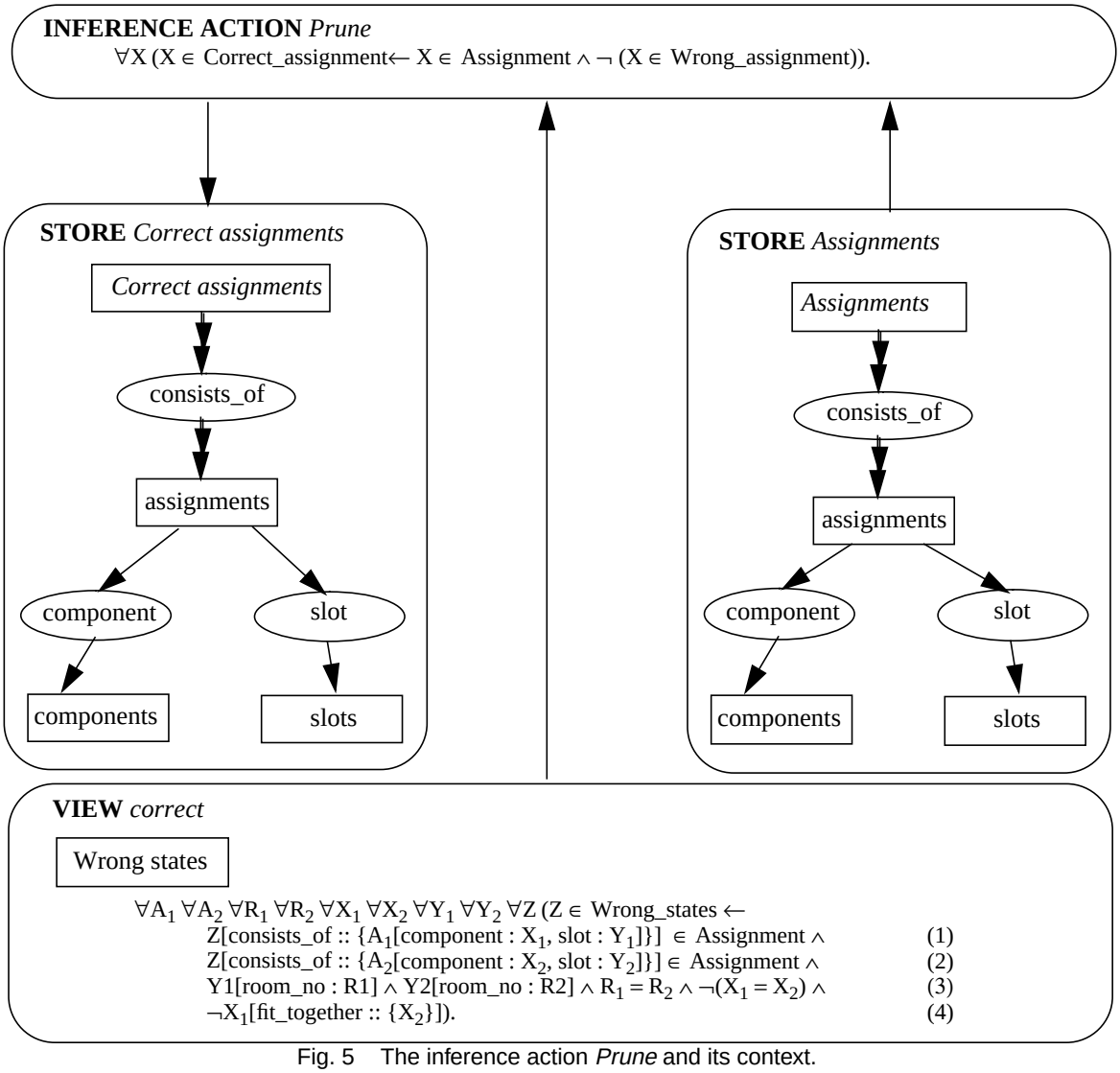

Fig. 5 The inference action *Prune* and its context.

A P-KARL program consists of elementary programs which are assignments. Composed programs are constructed from elementary ones by defining the control flow between these assignments. For this purpose, logical formulae can be used for defining 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.[2] 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* exists:

- (1) $(X_{k1},..., X_{kh}) := f_i(X_{j1},..., X_{jl})$
  f_i corresponds to an inference action and the $X_{k1},..., X_{kh}$ denote its output stores and terminators and the $X_{j1},..., X_{jl}$ its input stores;
- (2) $X_k := X_j$

2. Views are used to read domain knowledge and have no dynamic interpretation. They therefore do not appear at the task layer.

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[3];
- 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 defines of a loop that consists of successively creating assignments by extending them, pruning incorrect assignments, and checking whether a solution (i.e., a completed assignment) has been found (see Figure 3). The linear notation is given in Figure 6.

The syntax of P-KARL is just sugar on top of dynamic logic (cf. [Har84], [Koz90]) which defines the essence of the language. In the next section 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 for declaratively describing procedural programs.

```
WHILE ∅ solution(Solution)
DO
        Assignments := create (Assignments);
        Correct assignments := prune (Assignments);
        Assignments := Correct assignments;
        Check (Correct assignments);
ENDDO
```

Fig. 6 The task layer of the Sisyphus example defined by using the linear notation of P-KARL.

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

3 The Formal Semantics of KARL

In the following, we characterize the declarative semantics of the sublanguages L-KARL and P-KARL and finally the semantics of the complete language KARL. 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. The final subsection discusses how the mechanizability influenced design decisions of the declarative semantics. As KARL allows the representation of static and dynamic (i.e., procedural) knowledge, its semantics must include both types of knowledge. The development of KARL and its declarative semantics had to solve three problems: First, an object-oriented logic L-KARL was developed which can be used to specify static knowledge. L-KARL has the perfect Herbrand model semantics which is defined for Horn clauses with stratified negation. Second, dynamic logic was used to develop P-KARL for specifying knowledge about dynamics. P-KARL has the modal semantics of dynamic logic. Third, both languages had to be combined to represent a complete model of expertise. The semantics for this combination was reached by using the model-theoretical semantics of L-KARL for defining an interpretation for the P-KARL language. As a result, *the integrated description of static and dynamic knowledge based on a well-defined declarative framework* becomes possible.

During the following, we first present the essence of the different semantics and their combination. Then we present the different subjects (i.e., the semantics of L-KARL, P-KARL, and KARL) in more detail. Finally, we discuss issues related with trying to operationalize such a semantical framework.

3.1 A Sketch of the Different Semantics

The semantics is defined in three steps reflecting the internal structuring of KARL. First, we define the semantics of L-KARL. Because it is a syntactical variant of first-order logic we can do this in the same style as it is done for first-order logic. However, we have to provide a more complex definition of interpretations and models because of this syntactical extensions. For the Horn fragment of L-KARL we will define the perfect Herbrand model semantics. That is, we select for a set of Horn clauses one model from all possible ones as semantics. Second, we define the semantics of P-KARL. Because it is a subset of dynamic logic we can do this by q Kripke structure as done for dynamic logic. A state of a program is expressed by a value assigment of each variable and programs are defined as relations between such states (or worlds). Third, we have to define a semantics for a complete KARL specification including L-KARL and P-KARL of the different elements of a model of expertise in KARL. We achieve this by viewing the execution of an inference actions as elementary programs and interpreting the variables of dynamic logic by stores. We use the perfect Herbrand model of a set of clauses that define the inference plus the current content of a store according to the value assignment in dynamic logic to derive the new value assignment of the output store of the inference. That is,

$Y := f(X)$

becomes interpreted by the function f' that is defined by the perfect Herbrand model semantics of the clauses defining the inference action f'.

3.2 The Semantics of Logical-KARL (L-KARL)

Because L-KARL is a syntactical variant of first-order predicate logic, its semantics can be defined in two ways:

- We can introduce a mapping that transforms expressions in L-KARL to according expressions in first-order logic and applying the usual Herbrand model semantics for these expressions in first-order logic.
- We can provide a direct semantics for L-KARL by modifying and extending the model-theoretical semantics of predicate logic according to the extended syntax of L-KARL.

As done for F-logic (cf. [KLW95]) we have chosen the second way because it provides a *direct* semantics of a L-KARL specifications. Otherwise, one had to do a mapping into a lower-level language before assigning semantics to the modelling primitives. In that sense, first-order logic and L-KARL are related together like an assembler language and Pascal. The semantics can be expressed by mapping it on an assemebler language but it is usually more convinient to define a direct semantics. Actually defining the semantics by a lower-level language is usually necessary when executing a high-level language. We will see in section 4 that such a malling is actually done for executing KARL.

We start by defining an interpretation for a set of L-KARL formulae. Objects, classes, and values are interpreted by using 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. [KLW95]). For example, the following Horn clause collects all subclasses of class *c* as value of the set-valued attribute *subclass*:

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

The attributes and their domain and range restrictions are interpreted using *functions* on the universe. Specific conditions postulated for these functions realize multiple-attribute inheritance. The is-a and is-element-of hierarchies 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 an L-KARL language is a tuple $<U_E, U_\Sigma, U_V, \leq_U, I_U, I_{E\rightarrow}, I_{\Sigma\rightarrow}, I_{E\rightarrow\rightarrow}, I_{\Sigma\rightarrow\rightarrow}, I_{E\Rightarrow}, I_{\Sigma\Rightarrow}, I_{E\Rightarrow\Rightarrow}, I_{\Sigma\Rightarrow\Rightarrow}, I_\Pi, I_{A\Pi}>$ with:

- U_E is a subset of the domain for interpreting element denotations, U_Σ is a subset of the domain for interpreting class denotations, and U_V is a subset of the domain for interpreting values. The sets must be pairwise disjunct and their union *U* defines the *domain*.
- $\leq_U$ 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 using 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

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

$I_{\Sigma\to}$ and $I_{\Sigma\to\to}$ 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.

$I_{E\to}$, $I_{E\to\to}$, $I_{\Sigma\to}$, and $I_{\Sigma\to\to}$ 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 restrictions* are realized by $I_{E\Rightarrow}$, $I_{\Sigma\Rightarrow}$, $I_{E\Rightarrow\Rightarrow}$, and $I_{\Sigma\Rightarrow\Rightarrow}$ as proposed by [KLW95]. A partial function *f* on *U* is *anti-monotonic* if the fact that $v \leq_U u$ and $f(u)$ are defined implies 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 stricter 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 superclasses defining the same attribute. The subclass inherits all range restrictions of its superclasses. 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 to ensure that if *c* is used as a range restriction and $c \leq_U c'$ 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, we must define when an interpretation satisfies such a literal. Composed formulae which are constructed from these elementary formulae by logical connectives and quantifiers are interpreted in the usual manner. 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 its range restrictions.
- The value of the function which interprets the attribute *fit_together* is a superset 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 contains recursively a second data literal *dla[name : "Dieter*

Landes"], it is required that this data literal is also fulfilled by *I*.

An interpretation which fulfils a set of clauses is called a *model* of this set of clauses. Only a specific type of model is taken into account for L-KARL, in fact, only *Herbrand models* are admitted. This does not create a significant restriction because every set of formulae can be transformed into a logically 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 consequqence.* That is, if a ground positive literal ϕ follows from *H* then $\phi \in H$ must hold. A *congruence relation* on the Herbrand universe must be defined where each congruence class contains all syntactically different but semantically 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 the semantics for a set of L-KARL Horn clauses *S*. It is exactly the set of positive ground literals which are entailed by *S*. Negative literals are derived by 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 derived from a set of Horn clauses according to the minimal Herbrand model is larger than the set of negative literals which are derived from all models. Because Horn clauses are too restrictive, L-KARL also allows 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]). Considering only one model (i.e., the perfect Herbrand model) instead of all possible models makes the operationalization of KARL significantly easier.

3.3 The Semantics of Procedural-KARL (P-KARL)

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, one can distinguish different types of dynamic logic, e.g., propositional and first-order dynamic logic. In the following, the case of first-order formulae without quantifiers is discussed.

An interpretation provides a *domain* or *universe D*, some functions $\{f^A_1, f^A_2, ...\}$ used to interpret the function symbols and some relations on the domain $\{P^A_1, P^A_2, ...\}$ used to interpret the predicate symbols.

Simple atomic formulae have the form *P(X)* where *P* is a predicate symbol and *X* is a

variable. This formula is true for all states, where $\nu(X) \in P^A$. That is, where the value assignment of the variable X is an element of the relation which interprets the predicate symbol P. Complex formulae can be built up by logical connectives $\neg$, $\vee$, and $\wedge$.

A simple elementary programs looks like:

$y := f(x)$

It changes the value assignment of the variable y according to the current value of the variable x and the function used to interpret the function symbol f. The semantics of such an elementary program is the set of value assignments tuples where:

$(y := f(x))^A = \{(v,w) \mid v(z) = w(z) \text{ for } z \neq y \text{ and } w(y) = f^A(v(x))\}$

Complex programs can be built up from other programs and the following constructs:

- The *non-deterministic choice operator* $\cup$:
  $p \cup q$ means that the program can either execute p or q; and the semantics of $p \cup q$ is the union of the relations p^A and q^A which interpret p and q.
- The *sequence operator* ;:
  $p;q$ means that the program must execute p followed by q; and the semantics of $p;q$ is the composition of the relations p^A and q^A which interpret p and q.
- The *non-deterministic iteration operator* *:
  p^* means that the program could execute p a non-deterministic number of times, and the semantics of p^* is the transitive closure of the relation p^A which interprets p.
- The *test operator* ?:
  $\psi?$ is the set of all tuples (ν,ν) of states ν where ψ is true. That is, the execution path of the entire program is only continued for these states which evaluate ψ to true.

These operators can be used to model the control flow as it is known from procedural programming languages. For example, a program like

$\psi?;p \cup \neg\psi?;q$

models

IF ψ *THEN p ELSE q*

as it is known from procedural languages.

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, this restricts the expressibility of the language. Up until, these restrictions have not caused problems but it is clear that the design of our language immediately biases 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. In the next section we show how a L-KARL language is used to define such an interpretation.

3.4 The Semantics of KARL

In the following, we give the local semantics of an elementary inference action and describe its use for defining a global semantics of a complete KARL specification.

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. If it has an input store, then its semantics implicitly rely 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 *ia* is defined as the *perfect Herbrand model* of the union of

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

For example, the semantics of the inference action *prune* is the perfect Herbrand model of the current contents of the store *Assignments*, 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, then the inference action is not defined for the given input.

Having defined the semantics of an inference action, we now must define *how such an inference action changes the contents of its output stores*.[4] 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. Thus, a store can be used to select just the information which is needed out of the entire Herbrand model of an inference action. When an inference action uses a view, its perfect Herbrand model is a superset of the complete domain layer. Therefore, it would not make sense to store this complete model in a store. In addition, when an inference action has several output stores, the class definitions of the stores can be used to

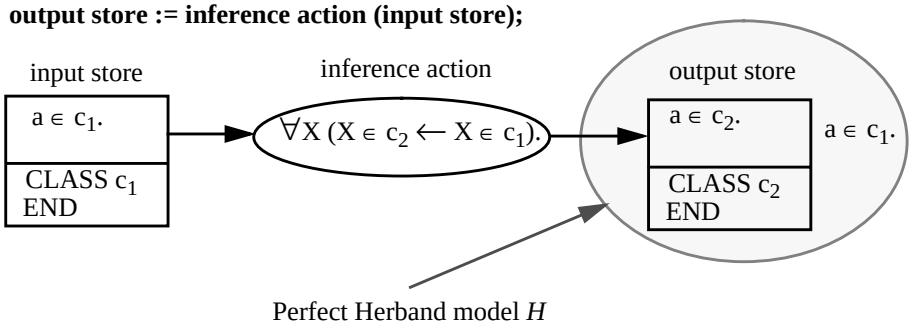

Fig. 7 An example of the semantics of an inference action.

4. We will not discuss the case in which a terminator is used to modify the domain layer because there is no technical difference.

choose different parts of the perfect model as their new contents.

For the new content of an store, only the is-element-of literals, equality literals, and data literals which fit to the class definitions of the store are selected. For example, an *is-element-of literal* $e \in c$ in H is only chosen if a class definition for c is contained by the store. Similarly, a data literal $e[...]$ in H is only chosen if an is-element-of literal $e \in c'$ exists as element of H with c' being contained by the store. Figure 7 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 which indicates that a is an element of c_1. The inference action is described by one clause which infers that every X is 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 $a \in c_1$ and $a \in c_2$ 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 which are contained by the model of the elementary inference action is employed as the new contents of an output store. This contents consist of *terms over the class definitions* of the store.

Def. 4 Terms over a set of class definitions[5]

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.

The Semantics of an Entire KARL Specification

An interpretation (D, $F^A = \{f_1^A,..., f_r^A\}$, $P^A = \{\varnothing_c, \varnothing_{c'},...\}$[6]) 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

5. In the paper, we simplify this definition. For more details see [Fen95b].
6. *c, c'*, etc. are the class names which occur in the class definitions of the stores.

Herbrand base and an element of its power set. A variable assignment assigns a set of ground literals to a variable. 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 6 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. That is, it should be true for all states, where *no* is-element-of literal $e \in c$ is assigned to the store X. In that way, it express the case, where a store is empty for the class c.

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 do not contain an is-element-of literal $e \in c$, with e is an arbitrary object denotation.

3.5 Mechanizability of the Declarative Semantics

A *declarative semantics* of a language A defines the meaning of an expression in A using expressions of a second (mathematical) language B. These definitions need not 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. These can be 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 is also intended to support prototyping, an *operational semantics* based on its declarative semantics has been defined. The operational semantics of KARL which will be introduced in the next section allows the derivation of output facts based on a given set of input facts which support the *validation* of a specification.

The declarative semantics of KARL has been developed with an eye to its operationalization. P-KARL programs are *regular* and *deterministic*. In addition, *recursion* is not available (i.e., the number of variables remains unchanged during execution). These restrictions significantly reduce the effort necessary for operationalization. In the non-deterministic case all states must be regarded simultaneously; or the interpreter must choose a state and the operationalization would thus be incomplete.[7] In the case of L-KARL, the language is restricted to a stratified set of clauses and only one specific model, the perfect Herbrand

7. Compare the discussion of [AEL+92], [LaL91] who operationalize loose specifications in VDM-SL.

model, is regarded to define the truth value of formulae. The restriction to one specific model significantly reduces the effort needed for operationalization. Normally, three different sets of ground positive literals exist in regard to a given set of formula *S*: literals which are entailed by *S*, literals which are entailed by *S* when they are negated, and literals which are neither a positive nor 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 to see whether the according positive literal is an element of the perfect Herbrand model. Work done in logic programming and deductive databases motivated our choice.

The main restriction of the operational semantics which will be introduced by the next section is the fact that user-defined classes and predicates (but not the defined value types like integer or string) have to have a *finite* extension. Otherwise, their minimal model would be infinite and therefore also its computation. Actually, this restriction has not yet caused problems in any application of KARL.

4 The Operationalization of KARL

In the following the main choices and reasons for the design of an interpreter for KARL are described. Firstly the different possibilities for the evaluation of L-KARL are discussed. Then the operational semantics for L-KARL and P-KARL are described. Finally the evaluation algorithms for L-KARL are sketched.

4.1 Selection of an evaluation strategy for L-KARL

The evaluation strategy of L-KARL can be discussed concerning two related but different issues. First, we discuss the distinction between *set-oriented* and *tuple-oriented evaluation*. The former computes all tuples which fulfil a predicate whereas the later computes only one of these tuples per time. Second, one can distinguish between bottom-up and top-down oriented evaluation. *Bottom-up evaluation* (also called *forward chaining*) starts from the facts and uses the rules to create new facts from the existing ones. *Top-down evaluation* (also called *backward chaining*) starts with the query, searches recursively for rules whose heads match the query and produces recursively new subqueries using the body atoms of the rules. Although in most cases set-orientation is combined with bottom-up techniques and tuple-orientation is combined with top-down evaluation these different aspects should not get mixed.

Set-oriented vs. one-tuple-at-a-time-oriented

One important decision for the design of an interpreter for KARL is the choice of an evaluation strategy for L-KARL. Such an evaluation strategy comes into action during the activation of an inference action. Such an inference action is described by a set of rules and facts. The effect of an inference action is to change the contents of its output stores. The content of an output store may be computed by posing several queries to the set of rules and facts of the inference action. For instance an output store contains only objects, which are elements of a class c of the output store. In Figure 7 it is shown, that only the fact $a \in c_2$ is an element of the output store; the fact $a \in c_1$ is not element of the output store, because the class c_1 is not defined for the output store. To determine which objects are elements of the class *c* the query:

$X \in c$

may be posed to the set of rules and facts. For every object identifier o which substitutes the variable X and thus makes this query true $o \in c$ is an element of the content of the output store. In a similar way the values of the attributes of such objects may be determined by posing adequate queries to the set of rules and facts.

One important parameter for the choice of an evaluation strategy is whether for a set of rules and facts and a query to this set all or only one solution is needed. According to the formal semantics both alternatives would be possible:

- If an evaluation strategy is used which produces only one answer for a query, the effect of the call of an inference action would be that for every class of every output store one element with all its attribute values is determined by posing a set of queries to the set of rules and facts. The selection which element out of the set of all possible elements may be done randomly. Using such an evaluation strategy the interpreter would work as an nondeterministic machine, because the result of the call of an inference action would be nondeterministic.
- If an evaluation strategy is used which produces all answers for a query, the effect of the call of an inference action would be that for every class of every output store every element with all its attributes is determined. So using this evaluation strategy the result of the call of an inference action is deterministic, because there is no need to choose one of the different possible solutions.

The first evaluation strategy (one-tuple-at-a-time) may be realized very efficiently using resolution-oriented top-down inferencing. For F-Logic, which is the basis of L-KARL a correct and complete proof-theory has been described which may easily be adapted to L-KARL. Otherwise this evaluation strategy has the severe disadvantage that the interpreter behaves in an nondeterministic way. This means that the interpreter behaves differently for the same test case in different runs. Additionally for one test case only one (randomly chosen) solution path is followed such that this test case may not be examined entirely. So the choice of such an evaluation strategy would contradict the needs for a prototyping environment, because prototyping needs the prototype to be validated sufficiently by testing. Finally, the specification cannot be used as gold standard for the implementation of the system which could deliver a different output which still could be correct.

The second evaluation strategy (set-oriented) does not provide these disadvantages. On the other hand evaluating all solutions of a posed query may be a very time consuming activity. In the area of deductive databases there has been a lot of effort to develop set-oriented evaluation strategies, which deliver all solutions to a posed query and which are much more efficient compared to using a one-tuple-at-a-time strategy to compute all solutions[8].

A set-oriented evaluation strategy may be realized much more efficiently than a one-tuple-at-a-time-oriented evaluation strategy because of the following reasons:

- For the single basic inferences set operations defined by the algebra of Codd may be used directly, which may be realized very efficiently. In [Ull88] it has been shown how inference steps may be mapped to expressions of Codd's algebra. For the computation of the conjunction of two atoms the join-operator must be used very frequently. The

8. In PROLOG this would be realized by artificially creating a failure of the proof in order to enforce the inference machine to backtrack and deliver the next solution.

join-operator needs $O(n \log(n))$ operations in the average. The computation of the conjunction by a one-tuple-at-a-time approach needs $O(n^2)$ operations [EIN89].

- Using a set-oriented approach duplicates may be eliminated. The detection of such duplicates is rather difficult and therefore time consuming within a one-tuple-at-a-time oriented approach.

Due to these advantages it has been decided to choose a set-oriented approach for the evaluation of sets of rules and facts in L-KARL.

Bottom-up vs. top-down evaluation

As already mentioned, *bottom-up evaluation* starts from the facts and uses the rules to create new facts from the existing oneswhereas *top-down evaluation* starts with the query searches recursively for rules whose heads match the query. For both strategies set-oriented or tuple-oriented versions exist.

The main advantage of a top-down oriented evaluation strategy is that it is more goal oriented. Pure bottom-up evaluation starts with the facts and computes all facts which are derivable by the rules. These new facts are again used as input for this process. Thus there are many facts computed where most of them do not affect the posed query. In most cases the facts which fulfil the query are only a small subset of the set of all computed facts. In contrast the top-down approach only considers those rules and facts which may be relevant to the posed query. Otherwise set-oriented evaluation can be much more efficiently achieved using a bottom-up approach because set-oriented top-down evaluation tends to produce a great number of smaller sets. So the efficiency of the set operations do not come to full effect (see [Ull89b]).

L-KARL includes equality reasoning. This means that if for two identifiers *a* and *b* it is stated that they denote the same object ($a = b$) then both must behave equally in the evaluation of the rules. If for instance the following facts and rules are given:

$a = b.$
$p(a).$
$q(c) \leftarrow p(b).$

Then $q(c)$ is a logical consequence of this set of rules and facts. Integrating equality reasoning into a top-down oriented evaluation strategy by *hyper-resolution* or similar techniques causes a large effort for evaluation. For a bottom-up approach only *matching*, i.e. unification of a ground term with a non-ground term, must be realized. This reduces the search space considerably.

Due to these reasons both—bottom-up evaluation and top-down evaluation—have advantages and disadvantages. Fortunately evaluation strategies have been developed which combine the advantages of both approaches (see [Ull89b]). For L-KARL an evaluation strategy based on such a combination has been developed.

4.2 The fixed-point semantics of L-KARL

L-KARL has a higher-order syntax. For instance, L-KARL allows sets of objects as values of attributes of an object. Nevertheless L-KARL has a direct first-order semantics. Therefore there exists a simple mapping from L-KARL to first order logic with equality. This mapping

allows an easier description of the operational semantics during the paper.

- Every atomic data literal $T_1[a:T_2]$ or $T_1[a::\{T_2\}]$ is mapped to an atom $a(T_1,T_2)$.
- Every P-literal $p(a_1:T_1,...,a_n:T_n)$ is mapped to an atom $p(T_1,...,T_n)$.
- Every is-a literal $T_1 \leq T_2$ is mapped to an atom $isa(T_1,T_2)$.
- Every is-element-of literal $T_1 \in T_2$ is mapped to an atom $el(T_1,T_2)$.
- Every equality literal $T_1 = T_2$ is mapped to an atom $eq(T_1,T_2)$.
- To integrate transitivity of the element relation the rule $el(t,c') \leftarrow el(t,c) \wedge isa(c,c')$ is inserted.
- To integrate transitivity of the is-a relation the rule $isa(c,c'') \leftarrow isa(c,c') \wedge isa(c',c'')$ is inserted.

Every ground instance of the binary predicate *eq* expresses that its two arguments are two different notations for the same object.

For the sake of simplicity, we do not consider negation and the original syntax of L-KARL instead we rely on its transformation to pure Horn logic with equality. For more details see [Ang93].

The characterisation of the minimal Herbrand model by a fixed point operator is the basis for the bottom-up evaluation of a set of rules. The fixed point operator is iteratively applied to the set of rules and the results of the previous application of the operator until a fixed point is reached. Then this fixed point represents the minimal Herbrand model.

Def. 8 (Fixed point operator)

Given a set K of rules of the form $A_0 \leftarrow A_1 \wedge ... \wedge A_n$, $n \geq 0$, where A_i, $0 \leq i \leq n$, are n_i - ary atoms of the form $A_i = q_i(b_{i1},...,b_{in_i})$. In the following definition I, I', I'' are sets of ground atoms without equality atoms and R, R' and R'' are equivalence relations on the set of all terms (representing the equality relation). $C(R)$ denotes the congruent closure of the equivalence relation R with respect to the functions in K. $I|_R$ denotes the quotient of I with respect to the congruence relation R. Be S a set of disjunct sets which divide the set I in the following way: every element $s \in S$, s is a maximal subset of I with the property that for any pair $p(c_1,...,c_n)$, $p(d_1,...,d_n)$ of elements in s, $(c_j,d_j) \in C(R)$ holds for all j, $1 \leq j \leq n$. $I|_R$ contains from every set in S exactly one element and it does not contain other elements. t_1 and t_2 are arbitrary terms.

The fixed point operator T_K is defined as follows:

$T_K((I,R)) = (I', R')$ with

$I'' := I \cup \{A_0\Theta \mid (A_0 \leftarrow A_1 \wedge ... \wedge A_n) \in K$, A_0 is not an equality atom and
for all i, $1 \leq i \leq n$, there exists a $q_k(a_{k1},...,a_{kn_i}) \in I$ and a ground substitution Θ, such that $(a_{kj},b_{ij}\Theta) \in C(R)$, for $1 \leq j \leq n_i\}$

$R'' := R \cup \{(t_1\Theta, t_2\Theta) \mid (eq(t_1,t_2) \leftarrow A_1 \wedge ... \wedge A_n) \in K$,
for all i, $1 \leq i \leq n$, there exists a $q_k(a_{k1},...,a_{kn_i}) \in I$ and a ground substitution Θ, such that $(a_{kj},b_{ij}\Theta) \in C(R)$, for $1 \leq j \leq n_i\}$

R' := the transitive, reflexive and symmetrical closure of R''

$I' := I''|_{C(R')}$

Whenever there is at least one function f in K and one equality $eq(a,b)$ then $C(R)$ contains (besides all reflexive equalities) an infinite set of equalities, because if $eq(a,b) \in R$ then $eq(f(a), f(b))$, $eq(f(f(a)),f(f(b)))$, ... are elements of $C(R)$.

The separation of the equality information in the equivalence relation R resp. R' from the other facts in I resp. I' and the application of the quotient operator to the set I'' is not necessary to define a fixed point operator for Horn logic with equality (see [SpA92]) but it allows to develop a much more efficient evaluation strategy.

Def. 9 (Fixed point)

The least fixed point FP_K of a set of rules and facts K is defined by

$FP_K := T_K^{\infty}((\{\},\{(t,t) \mid t \text{ is an element of the Herbrand universe}\}))$

In [Ang93] it has been shown that the least fixed point for a set of L-KARL rules always exists and that this least fixed point can easily be mapped to the minimal (i.e., perfect) Herbrand model which constitutes the semantics of a set of L-KARL rules.

In order to operationalize this fixed point operation for an KARL interpreter the fixed point has to be finite. Infinite recursions of rules, unsafe rules, the application of built-in predicates within recursion and the equality predicate are reasons why the fixed point may become infinite. In [Ang93] restrictions for the rules are described which enforce the finiteness of the resulting fixed point. These restrictions are syntactical and are thus statically provable.

4.3 The Operational Semantics of a KARL Program

The semantics of an entire KARL program is defined by successively transforming one program state to the next state by the mapping defined by the activated inference action. So every statement at the task layer either changes the program state which is defined by the contents of the stores and the program counter or it only changes the program counter. This semantics is quite similar to the semantics of deterministic while-programs (see [LoS84]).

The state of a program is defined by the contents of the stores:

Def. 10 (state)

A variable assignment is a function which maps each variable (store) to a subset of the Herbrand base and each boolean variable to one of the values *true* or *false*. The state of a program is one variable assignment of all variables.

A configuration of a KARL program represents the whole process state. It consists of the state and additionally the remaining program which has to be processed.

Def. 11 (configuration)

A configuration is a pair (w,σ), where w is the remaining to be processed program and σ is the state of the program (an assignment for all variables).

The following transition relation describes for each statement at the task layer how it changes the process state, i.e. how it relates a configuration with its succeeding configuration. This transition relation is described for two types of statements available at the task layer.

Def. 12 (transition relation)

Given the two configurations (w_1,σ_1) and (w_2,σ_2), the transition relation $\Rightarrow$ is defined by:

$(w_1,\sigma_1) \Rightarrow (w_2,\sigma_2)$

with

1. $w_1 = (X_{k1},...,X_{kh}):=f_i(X_{j1},...,X_{jl})$; w_2,
   w_1 is the call of an inference action and σ_2 is the new variable assignment (with changed contents of the output stores of the inference action f_i).
2. w_1 = IF B THEN w_1' ELSE w_2' ENDIF; w_3'
   $\sigma_2 = \sigma_1$
   and $w_2 = w_1'; w_3'$, if B is true for σ_1 or $w_2 = w_2'; w_3'$, if B is false for σ_1

The semantics of the KARL program is defined by the function $M(P)$, which maps the start configuration (P, σ_0) to the final configuration (λ, σ_k), where λ denotes the empty program. This mapping exists if there exists a sequence of configurations where two succeeding configurations are related by the transition relation, the first configuration of this sequence is the start configuration (P, σ_0) and the last configuration is the final configuration (λ, σ_k). If there does not exist such a sequence (the program does not stop) then the semantics is undefined.

4.4 The evaluation method DFE

In the following section the evaluation algorithm DFE for sets of L-KARL rules and facts is sketched. DFE stands for Dynamic Filtering with Equality.

System Graph

The evaluation algorithm works on a data structure called system graph [KiL86]. This graph represents the set of rules. Every atom of the rules is represented as a vertice of the graph. All atoms in the body of a rule are connected to the head atom of the rule. A head atom is connected to all body atoms with the same predicate symbol. Let's have a look at an example.

Example 1

Given the following set of rules

$1.\forall X\forall Y\ (eq(X,Y) \leftarrow p(X,Y) \wedge q(Y)).$
$2.\forall X\forall Y\ (r(X,Y) \leftarrow p(X,Y) \wedge r(Y,b)).$
$3.\forall X\forall Y\ (r(X,Y) \leftarrow s(X,Y)).$
$q(a).\ p(b,a).\ p(c,b).\ r(a,a).\ r(c,b).\ s(e,f).$

and a query

$\forall Y\ (\leftarrow r(a,Y)).$

The corresponding system graph is shown in Figure 8.

Data propagation and query evaluation using the system graph

The bottom-up evaluation using the system graph may be seen as a flow of data from the sources to the sinks along the vertices of the graph.

If a fact $q(a_1,...,a_n)$ flows from a head atom of rule r to a body atom $q(b_1,...,b_n)$ of rule r' (along a solid arrow) a match operation takes place. This means that the non-ground body atom has to be unified with the facts produced by rule r. As an extension of the match-operation in [KiL86] this operation has to take the equalities into account. All substitutions for a body atom form the tuples of a relation which is asserted to the body atom. Every tuple of this relation provides a ground term for every variable in the body atom. To evaluate the rule all relations of the body atoms are joined and the resulting relation is used to produce a set of new facts for the head atom. These facts again flow upwards in the system graph.

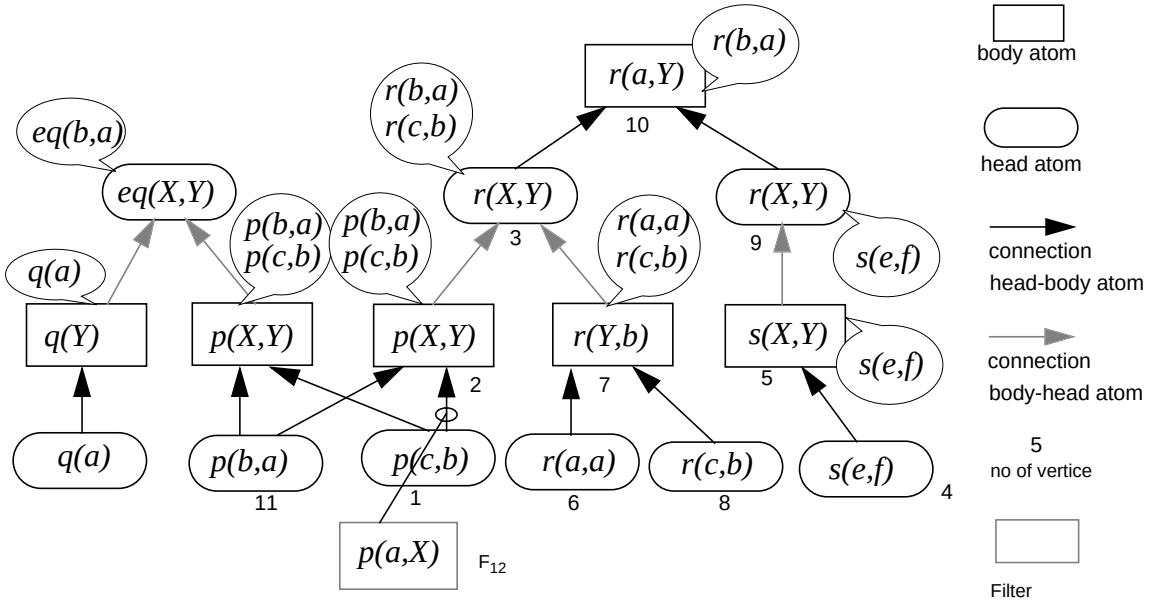

Fig. 9 Bottom-up evaluation.

This evaluation strategy corresponds to naive evaluation [Ull88] and realises directly the above mentioned fixed point operator. Because the system graph may contain cycles (in case of recursion within the set of rules) semi naive evaluation [Ull88] may be applied to increase efficiency.

The bottom-up evaluation of the example mentioned above is shown in Figure 9. The facts flowing to a vertice are shown in the bubble attached to the vertice.

Note that for the flow of the fact *r*(*a*,*a*) to the vertice of the body atom *r*(*Y*,*b*) the fact that term *b* is equal to term *a* has to be known. This means that this equality has to be determined before the fact *r*(*a*,*a*) can flow to the body atom *r*(*Y*,*b*). This equality has also to be considered in creating the fact *r*(*c*,*b*) at the vertice of the head atom *r*(*X*,*Y*) from the conjunction of the facts *p*(*c*,*b*) and *r*(*a*,*a*) of the body atoms *p*(*X*,*Y*) and *r*(*Y*,*b*).

The result of an evaluation corresponds to the entire perfect Herbrand model of a set of rules and facts. But to determine the contents of the output stores of an inference action normally only a proper subset of this model is needed (see Figure 7).

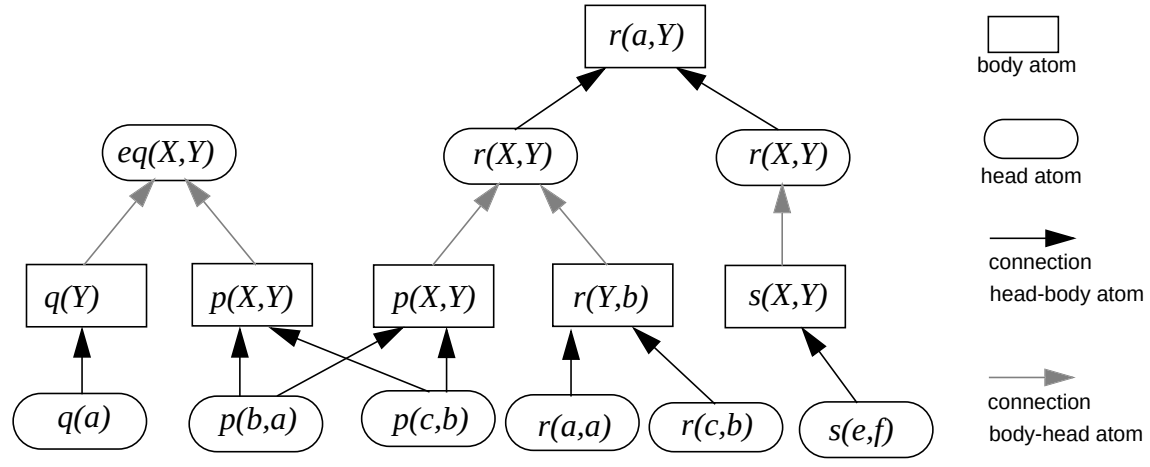

Fig. 8 System graph for the example.

In order to reduce the evaluation effort, i.e. to evaluate only a small superset of the needed subset of the perfect Herbrand model the evaluation algorithm has been optimized by combining it with a top-down strategy.

Optimization of the data flow

Figure 9 exemplifies that facts are created for the vertices although they are useless for answering the posed query. For instance the fact $p(c,b)$ flowing to the body vertice $p(X,Y)$ cannot contribute in any way to the answer, because of the query $r(a,Y)$ only instances of the variable X which are equal to the constant a (a or b) are relevant and c is not equal to a. Such useless facts then in turn produce new useless facts in the subsequent evaluation. The key idea of the *dynamic filtering technique* is to abort the flow of useless facts as early as possible (i.e. as close to the sources of the graph as possible) attaching so-called filters to the edges of the graph. Such a filter consists of a set of atoms. A filter lets a fact pass through if there exists an atom within the filter which matches with the fact.

For instance the filter F_{12} between vertice 1 and vertice 2, $F_{12} = \{p(a,X)\}$ prevents the fact $p(c,b)$ from flowing to the vertice 2 because no ground substitution ζ exists such that $(a\zeta,c) \in C(R)$ (in the example only a and b are derived as equal). Additionally the creation of the fact $r(c,b)$ for vertice 3 is prevented. Thus the answer to the posed query $r(a,Y)$ remains the same, although the amount of facts flowing through the graph is reduced.

The filters at the edges of the system graph are created by propagating constants within the query, within the rules, or within already evaluated facts downwards in the graph. For instance the filter $F_{12} = \{p(a,X)\}$ is determined using the constant a at the first argument position of the query $r(a,Y)$. This filter is valid because for the answer only facts at vertice 3 are useful containing an a or a constant b with $(a,b) \in C(R)$ (a and b are equal) as first argument. So variable X in rule 2 must be instantiated with a or a constant b equal to a only in order to be useful for the query.

The filters at the edges of the system graph are created during the evaluation process in the following way. Assume that all equalities are known in advance. First of all constants within the query and within the rules are propagated downwards in the graph. Starting at the query or at a body atom they are propagated to all head atoms which are connected to this atom. From the head atoms they are propagated to the first body atom of the corresponding rule and from there in the same way downwards. In propagating the constants downwards they produce new filter atoms for the filters at the edges between the head atoms and the body atoms. Now new facts may flow through these newly created (or modified filters). The constants within facts which have flown to a body atom are now propagated sideways to the next body atom of the same rule. From there they are again propagated downwards and thus produce new filter atoms which again allow new facts to pass. This process proceeds until no more facts flow upwards and no more filter atoms are created. The posed query is answered now by all facts which have flown to this query.

Let us demonstrate this at our running example. The query atom is propagated downwards. This creates the filters $F_{9,10} = \{r(a,Y)\}$, $F_{4,5} = \{s(a,Y)\}$, $F_{3,10} = \{r(a,Y)\}$, $F_{11,2} = \{p(a,Y)\}$ and $F_{1,2} = \{p(a,Y)\}$. Because a and b are equal the fact $p(b,a)$ can flow through $F_{11,2}$. By sideways propagation at vertice 7 the instance $r(a,b)$ is created which again is propagated downwards. This leads to the filters $F_{6,7} = \{r(a,b)\}$ and $F_{8,7} = \{r(a,b)\}$. Because a and b are equal the fact $r(a,a)$ can flow through $F_{6,7}$ to vertice 7. With the fact $p(b,a)$ which has flown

to vertice 2 a new instance $r(b,a)$ of the head atom is created. This fact flows through $F_{3,10}$ to vertice 10 and answers the query $(Y = a)$.

This mechanism works fine if all relevant equalities are known in advance. In the above described course of events it is essential to know the equivalence of *a* and *b*. But knowing in advance means that all equalities have to be determined in advance by a bottom-up evaluation of all rules which may contribute to the determination of equalities. This may be a large effort and additionally requires finite extensions of the corresponding head atoms. These disadvantages may be avoided by determining only those equalities that are actually needed.

At the beginning of the evaluation all equalities for the constants *a* within the rules and the query are determined by posing the queries $\leftarrow eq(a,X)$ and $\leftarrow eq(X,a)$ to the set of rules and facts and evaluating these queries using the above mentioned algorithm. After this evaluation all terms equal to *a* are known.

During the evaluation process, three different events may lead to the derivation of new terms from existing ones (for which the equalities have to be determined):

- When new facts flow to a body atom of a rule at the leaves of the graph, new terms come into play. For instance if the fact $p(b,a)$ flows through $F_{11,2}$ the new constant *b* comes into play.
- At the point where a rule is evaluated new facts for the head atom are created which may contain new terms. For instance a rule $p(f(X)) \leftarrow q(X)$ creates a new term $f(a)$ for the instance *a* of *X*.
- When constants are propagated sideways, the instantiation of the body atom may create new terms. For instance for rule $p(Y) \leftarrow q(X,Y) \wedge r(f(X))$ a substitution $\sigma = \{X/a, Y/b\}$ creates the instance $r(f(a))$ of the next body atom and thus creates a new term $f(a)$.

For these new terms the equalities have to be known in order to propagate the facts upwards and downwards in the graph correctly. For newly created ground terms *t* as well as for their subterms the queries $\leftarrow eq(X,t)$ and $\leftarrow eq(t,X)$ are posed to the set of rules while the evaluation process is running. Thus in order to evaluate a query additional queries may be created in order to derive the necessary equalities. The new queries additionally weaken the filters and let thus additional facts flow in the graph. If a new equality $eq(a,b)$ has been evaluated (a corresponding fact has flown to an equality vertice) it has to be checked for those facts which contain *a* or *b* whether they now flow through filters or are now matched by a body atom or create now new filter atoms in propagating downwards. By this way for a query only those equalities are determined which may contribute to the answer of the query. Thus only a small subset of all equalities have to be evaluated normally which may reduce the evaluation effort considerabely compared to the evaluation of all equalities in advance.

The above described algorithm combines the advantages of set-oriented evaluation, bottom-up evaluation and top-down evaluation. The advantages of set-oriented evaluation are gained by realising the join-operations efficiently in time $O(n\log n)$. Bottom-up evaluation provides larger, but fewer sets of facts, reduces the additional search space caused by the equalities and avoids infinite branches in the proof tree. The advantages of top-down evaluation are gained by the propagation of constants within the graph which leads to the evaluation of a subset of the perfect Herbrand model only.

5 The Tool Environment of KARL

Based on the operational semantics for KARL and on the evaluation algorithm DFE for sets of L-KARL rules and facts an interpreter for KARL has been developed. The interpreter integrates a lot of debugging facilities:

- The graphical interface shows the model of expertise using the graphical representation of KARL and supports the refinement facilities at the inference and task layer.
- The contents of stores may be inspected and modified.
- The problem solving process may be traced step by step.
- Breakpoints may be defined.
- The execution is possible in both directions, i.e. the debugger can run one step backwards for example.
- The whole state of the debugger may be saved and loaded.

As the formal and executable specification in KARL uses the KADS model of expertise as its framework the validation of the specification become possible in terms of the conceptual model which is also used to specify an informal model of the system. This buys two important advantages. First, a smooth transition from informal and formal as well as operational specifications become possible (see [FeN93]). Second, the understandability of the formal specification, of the validation process and of its results is significantly improved for the system developer as well as for the user as they can interpret and communicate at the conceptual level.

Figure 10 shows a screen dump of the graphical interface for the debugger. It shows the inference structure of the model of expertise. The content of a store may be inspected by clicking on the symbol of the store on the screen.

This debugger allows to observe the running problem solving process step by step and thus allows to validate it by testing. This validation provides the necessary feed back for the knowledge engineer and the expert in order to develop an adequate model of the problem solving process by explorative prototyping.

There exists an environment called MEMO-Kit ([Neu93], [NeM93]) which supports earlier phases in the development process of a model of expertise. MEMO-Kit offers editors to create and modify informal models and semiformal models using graphical primitives. These models are transformable to KARL models. MEMO-Kit allows to represent the model of expertise in a semiformal manner and thus supports the communication process between the knowledge engineer and the expert.

6 Applications

KARL has already been applied and evaluated in different case studies which resulted either in a formal model of expertise for an expert system application or in a formalization of a problem solving method.

KARL Models of Expertise for Expert System Applications

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

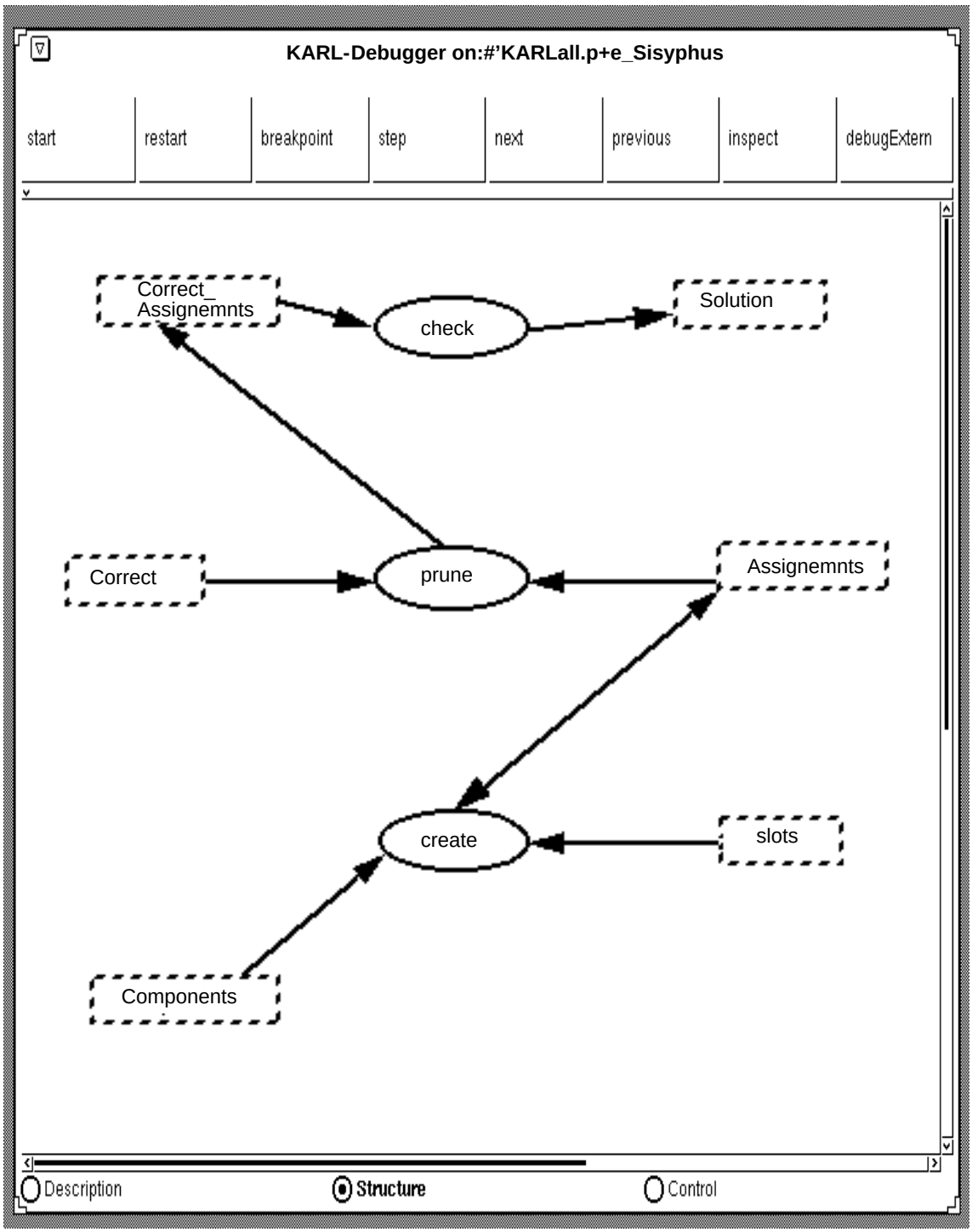

Fig. 10 Screen dump of the debugger of KARL.

paper. Among others the following applications have been tackled:

- Room Assignment Task (Sisyphus-1) [Lin92]: The Sisyphus-1 task consists of assigning a set of employees to appropriate office rooms meeting a predefined set of constraints. In [AFL+92a] and [AFL92c] two solutions for the Sisyphus-1 problem are described providing a complete model of expertise specification.

- Scheduling Task: 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.
- Selection of Scheduling Algorithms: Selecting operations research algorithms, which are suitable for solving a given application task, from a large library is a significant problem. In [KFG92] an approach for solving the problem of selecting appropriate scheduling algorithms from a library in a given project management context is discussed.
- Elevator Configuration Task (Sisyphus-2/VT): As a second problem for comparing different knowledge level modeling approaches and languages an elevator configuration task has been posed [Yos92]. In [PFL+96] a KARL model of expertise for solving the VT task is described. The detailed specification of such a complex task gave us various insights in the advantages and disadvantages of some of the design decisions which we have made when developing KARL.

Specification of Problem Solving Methods in KARL

We also specified some problem solving methods with KARL. Problem-solving methods can be seen as a model of expertise with an empty domain layer. They define a generic reasoning strategy that should be applicable to a complete class of tasks and domains:

- Weak Problem Solving Methods: The development of several solutions for the Sisyphus-1 task (room assignment) resulted in the specification of the weak problem solving methods *hill climbing* [AFL+92a], *chronological backtracking* [FEM+96], and *beam search* [FEM+96].
- Cover and differentiate: In [Ang92] a complete specification of the strong 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.
- Board-game method: 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+96]. This case study is a typical example how KARL can be used for re-engineering already implemented problem solving methods.
- Propose and exchange: The strong problem solving method *propose and exchange* [PoP92] has been formalized for solving a simple design task [LFA93]. This problem solving method is well-suited for solving assignment problems.
- Propose and revise: In the context of the Sisyphus-2 VT task a formal specification of the strong problem solving method *propose and revise* has been developed [PFL+96]. This problems solving method can be used for solving problems where specific revise rules exist for solving constraint violations.

7 Comparison with Related Languages

In the following, we will briefly compare KARL with specification languages from the field of knowledge engineering ($(ML)^2$ and DESIRE), information systems development (Telos and TROLL), and software engineering (VDM and Z). A more detailed comparison of KARL with most existing knowledge specification languages can be found in [FeH94], [AFL92b] contains a comparison of KARL with a specification language for developing information systems and [Fen95c] provides a comparision with languages stemming from software engineering.

The main difference between KARL and *$(ML)^2$* [HaB92] is based 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 in order 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 would be less efficient than an operationalization of Horn logic as used by KARL. 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, epistemologically different types of knowledge like concepts, attributes, domain restrictions and range restrictions are uniformly represented by predicates in $(ML)^2$. Both languages use dynamic logic for specifying procedural knowledge. On the other hand, 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 action is represented by a predicate and the logical description of the predicate is integrated into dynamic logic through 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 a less expressive variant of dynamic logic is sufficient for P-KARL because every formula only contains free variables (cf. [Koz90]). *DESIRE* [LPT93a] is also a knowledge specification language but differs significantly from KARL in the conceptual model it applies as DESIRE does not rely on the KADS model of expertise. The main difference is that DESIRE uses a 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 using 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 for reasoning about composition of actions like sequences or loops.

Telos [MBJ+93], which has evolved from RML (Requirements Modeling Language, see [GMB94]) is a language for supporting the development of information systems. It views the specification of an information system as an object-oriented knowledge base supplemented by constraints and rules. An interval-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. In addition to 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 the two 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 for requirements specification or conceptual modeling—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 behavorial aspects. A basic characteristic of TROLL is the use of temporal logic for specifying, e.g. the admissible state sequences of objects. The formal semantics of TROLL is given in [Jun93] using the so-called Object Specification Logic [SSC92]. However, a complete operationalization of TROLL is not available. Again, a major difference between KARL and TROLL is the fact that the latter aims primarily at the specification of objects and their behavior. The behavior is characterized by rather simple operations which change the state of the corresponding objects and may cause a change in the state 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 TROLL. On the other hand, it should be clear that, e.g., ideas 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 TROLL constrains possible control flows by using temporal logic, KARL uses dynamic logic to explicitly define the control flow in a procedural manner. As already mentioned, this design decision in KARL was made because experience in large expert system projects like XCON (cf. [SBJ87], [BaS89]) have shown that great problems arise if the control flow is only implicitly specified.

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, in fact application systems which are developed with a classical software engineering techniques like structured analysis can be formally specified in KARL as well. On the other hand, there are no means to specify real-time problems in KARL.

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 means of 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. Preconditions, post-conditions, and invariants are means for specifying these data models and operations. VDM provides a number of proof obligations which can be used to show the mathematical self-consistency of a specification (cf. [BFL+94] for an introduction to proofs in VDM-SL). *Z* is based on typed set theory. Static and dynamic aspects of a system are uniformly described by so-called schemes. Complex specifications can be built up by combining several of these schemes. A schema describes a data type or an operation by means of preconditions, postconditions, and invariants. For this purpose, sets, relations, and functions can be defined using a language similar to predicate logic. [Spi92] defines a standard for Z and its mathematical semantics is defined in [Spi88].

A difference when comparing the languages *VDM* and *Z* with KARL is that the latter is generally subject to a stronger conceptual model of the system to be described. KARL uses the KADS model of expertise and is thus much closer to a conceptual and informal or semi-

formal description of expertise than general purpose languages like *Z*, which describe arbitrary programs with the help of mathematical set theory. On the other hand, as a result of the effort to put VDM and Z into practice, several authors developed combinations with semi-formal specification techniques like structured analysis [You89] or object-oriented analysis [CoY91]. Approaches like [ELP93], [LPT93b] for VDM, [Ran90], [SBC92] for Z (or [FrD89] for algebraic techniques) are therefore in this respect close in spirit to KARL.

In addition, VDM and Z were designed as formal specification languages. Supporting the specification process by prototyping with executable specifications as supported by KARL was not a goal of these projects. More recent approaches to VDM developed interpreters for subsets of the language (cf. [AEL+92], [ELL94], [LaL91]). Since the full VDM-SL language is not executable in general, these interpreters have to exclude language features like infinite sets, execution of type binding, purely implicitly defined functions and operations. A further problem for the operationalization effort is caused by loose specification.[9]

Finally, a difference between KARL and specification languages of Software Engineering lies in the fact that these languages aim for a declarative specification of the functionality of a system. They try to abstract from *how* this functionality is achieved. As already mentioned, in knowledge-based system development of part of the “how” is regarded as essential expertise which must therefore be specified. Therefore, an inference layer in KARL specifies the significant inferences of a problem-solving process and the task layer supplements these definitions with a control flow which should ensure an effective and efficient computation of a solution. KARL specifies in an abstract manner how a solution is achieved instead of only describing *what* the solution is.

8 Conclusion

In this paper a detailed specification as well as a design rationale of the Knowledge Acquisition and Representation Language KARL are given. In essence, the development of KARL aimed at fulfilling the following requirements:

- KARL should be based on a conceptual model which provides means for specifying the functionality of a knowledge-based system in an implementation independent way. For that purpose, KARL uses the well-known KADS model of expertise as the underlying conceptual model. Thus, different types of knowledge are explicitly distinguished and specified separately.
- KARL should be a formal and executable specification language. During the process of defining the KARL language it became clear that a lot of well-balanced design decisions have to be made to end up with a language which is both, a formal and an executable language. Furthermore, for having a precisely defined implementation basis available a formal operational semantics had to be developed supplementing the declarative formal semantics.
- KARL should be defined in a way that improves the comprehensibility of KARL specifications. Therefore, most KARL primitives have a graphical representation. All the case studies which have been carried out so far indicate that the graphic representation of KARL specifications considerably enhances their comprehensibility.

9. Compare [HaJ89] and [Fuc92] for a discussion on pro and cons about executable specification languages.

When considering KARL from a technical point of view it becomes apparent that the design of KARL is influenced by methods from knowledge engineering, software engineering, and deductive data bases. In that way, KARL is a good example of the close relationships which exist between these different research areas.

Some shortcomings of KARL have been detected during its applications. First, Horn clauses often enforce an artificial manner in modeling problems. Therefore, syntactical extensions of Horn logic as proposed by [LlT84] should be included in a revised version of KARL. Second, the mentioned VT-task showed the need of an object-meta-relationship between domain and inference layer. Rules of the domain layer must be treated as objects at the inference layer. For example, one must determine which is the appropriate repair rule for a violated constraint. We could bypass this problem be modeling such repair rules twice. Once as objects and once as rules. In addition, these rules contained a premise which turn them on and off. So it was doable but not at all in a nice and natural manner. Third, the deterministic control at the task layer enforces over-specification in cases where no complete and deterministic control flow should be specified. Otherwise, prototyping of non-deterministic specifications causes serios problems (cf. [AEL+92], [LaL91]).

Acknowledgement

We want to thank Dieter Landes for many helpful discussions concerning the development of KARL, Susanne Neubert for her work on MEMO-Kit, Robert Lechler and Michael Wagner for implementing the KARL interpreter, and Manfred Aben, Frank van Harmelen, Jan Treur and two anaonymous reviewer for their comments on drafts of the paper

Bibliography

[AEL+92] M. Andersen, R. ElmstrØm, P. B. Lassen, and P. G. Larsen: Making Specifications Executable—Using IPTES Meta-IV. In *Microprocessing and Microprogramming*, vol 35, September 1992.

[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.

[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), PhD thesis, Infix-Verlag, 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.

[Bar93] M. Barbuceanu: Models: Toward Integrated Knowledge Modeling Environments. In *Knowledge Acquisition*, vol 5, 1993.

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

[Bee90] C. Beeri: A Formal Approach to Object-Oriented Databases, *Data & Knowledge Engineering*, vol 5, no 4, 1990, pp. 353-382.

[Ben95] R. Benjamins: Problem Solving Methods for Diagnosis And Their Role in Knowledge Acquisition,

International Journal of Expert Systems: Research and Application, 8(2):93—120, 1995.

[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.

[Bra79] R. J. Brachman: On the Epistemological Status of Semantic Networks. In N. V. Findler (ed.), *Associative Networks: Representation and Use of Knowledge by Computers*, Academic Press, New York, 1979.

[BvV94] J. Breuker and W. Van de Velde (eds.): *The CommonKADS Library for Expertise Modelling*, IOS Press, Amsterdam, The Netherlands, 1994.

[CoY91] P. Coad and E. Yourdon: *Object-Oriented Analysis*, 2nd ed., Yourdon Press, Englewood Cliffs, 1991.

[ElN89] R. Elmasri and S.B. Navathe: *Fundamentals of Database Systems*, The Benjamin/Cummings Publishing Company, Houston, 1989.

[ELL94] R. ElmstrØm, P. B. Lassen, and P. G. Larsen: The IFAD VDM-SL Toolbox: A Practical Approach to Formal Specifications. In *ACM SIGPLAN Notices*, summer 1994.

[ELP93] R. ElmstrØm, R. Lintulampi, and Mauro Pezze: Giving Semantics to SA/RT by Means of High Level Timed Petri Nets. In *Real-Time Systems*, vol 5, no 2-3, May1993.

[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.

[FAL93] D. Fensel, J. Angele, D. Landes, and R. Studer: Giving Structured Analysis Techniques a Formal and Operational Semantics with KARL.

[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+96] DD. Fensel, H. Eriksson, M. A. Musen, and R. Studer: Developing Problem-Solving by Introducing Ontological Commitments, *International Journal of Expert Systems: Research & Applications*, 9(4), 1996.

[Fen95a] D. Fensel: Assumptions and Limitations of a Problem-Solving Method: A Case Study. In *Proceedings of the 9th Banff Knowledge Acquisition for Knowledge-Based System Workshop (KAW´95)*, Banff, Canada, February 26th - February 3th, 1995.

[Fen95b] D. Fensel: *The Knowledge Acquisition and Representation Language KARL*, Kluwer Academic Publ., Boston, 1995.

[Fen95c] D. Fensel: Formal Specification Languages in Knowledge and Software Engineering, *The Knowledge Engineering Review*, 10(4), 1995.

[FeN93] D. Fensel and S. Neubert: Integration Of Semiformal and Formal Methods For Specification of Knowledge-Based Systems. In *Proceedings of the GI-Fachgespräch Workshop-F1 Integration of Semiformal and Formal Methods, IFIP´94*, Hamburg, August 31, 1994.

[Flo84] C. Floyd: A Systematic Look at Prototyping. In R. Budde et al. (eds.), *Approaches to Prototyping*, Springer-Verlag, Berlin, 1984, pp. 1-18.

[FrD89] R. B. France and T. W. G. Docker: Formal Specifications Using Structured System Analysis. *In Proceedings of the 2nd European Software Engineering Conference ESEC´89*, Warwick, September 11-15, Lecture Notes in Computer Science, no 387, Springer-Verlag, Berlin, 1989.

[Fuc92] N. E. Fuchs: Specifications Are (Preferably) Executable. In Software Engineering Journal, vol 7, September 1992.

[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.

[HaJ89] I. J. Hayes and C. B. Jones: Specifications are not (necessarily) Executable. In Software Engineering Journal, vol 4, no 6, November 1989.

[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.

[KiL86] M. Kifer, E. Lozinskii: A Framework for an Efficient Implementation of Deductive Databases. In *Proceedings of the 6th Advanced Database Symposium*, Tokyo, 29.-30. August 1986, 109-116.

[Kim90] W. Kim: *Introduction to Object-Oriented Databases*, The MIT Press, Cambridge, Massachusetts, 1990.

[KiW89] M. Kifer and J. Wu: A Logic for Object-Oriented Logic Programming (Maier´s O-Logic Revisited). In *ACM Symposium on Principles of Database Systems*, Philadelphia, March 29-31, 1989, pp. 379-393.

[KLW95] M. Kifer, G. Lausen, and J. Wu: Logical Foundations of Object-Oriented and Frame-Based Languages, Journal of the ACM, 42:741—843, 1995.

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

[LaL91] P. B. Lassen and P. G. Larsen: An Executable Subset of Meta-IV with Loose Specification. In *Proceedings of the VDM´91 Formal Software Development Methods*, Noordwijkerhout, The Netherlands, Oktober 1991, Springer-Verlag, Berlin, 1991.

[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].

[LlT84] J. W. Lloyd and R. W. Topor: Making Prolog More Expressive, *Journal of Logic Programming*, vol 1, no 3, 1984.

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

[LoS84] J. Loecks, K. Sieber: *The foundations of program verification*, Wiley, Teubner, Stuttgart, 1984.

[LPT93a] I. van Langevelde, A. Philipsen, and J. Treur: A Compositional Architecture for Simple Design Formally Specified in DESIRE. In [TrW93].

[LPT93b] P. G. Larsen, N. Plat, and H. Toetenel: A Formal Semantics of Data Flow Diagrams. In *Formal Aspects of Computing*, vol 3, 1993.

[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.

[PFL+96] K. Poeck, D. Fensel, D. Landes, and J. Angele: Combining KARL and Configurable Role Limiting Methods for Configuring Elevator Systems. Combining KARL And CRLM For Designing Vertical Transportation Systems, *International Journal of Human-Computer Studies (IJHCS)*, 44(3-4), 1996.

[PoP92] K. Poeck and F. Puppe: COKE: Efficient Solving of Complex Assignment Problems with the Propose-And-Exchange Method. In *Proceedings of the 5th International Conference on Tools with Artificial Intelligence*, Arlington, Virginia, November 10-13, 1992.

[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.

[Ran90] G. P. Randell: *Translating Data Flow Diagrams into Z (and Vice Versa)*. Technical Report 90019, Procurement Executive, Ministry of Defence, RSRE, Malvern, Worcestershire, UK, October 1990.

[SBC92] S. Stepney, R. Barden, and D. Cooper (eds.): *Object Orientation in Z*, Springer-Verlag, Berlin, 1992.

[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.

[SpA92] V. Sperschneider, G. Antoniou: *LOGIC: A Foundation for Computer Science*, International Computer Science Series, 1992.

[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.

[Ull89] J. Ullman: Bottom-up beats top-down for Datalog. In *Proceedings of the 8th ACM Symposium on Principles of Database Systems (PODS)*, Philadelphia, USA, 1989.

[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.