The Knowledge Acquisition and Representation Language KARL

!! DRAFT ONLY !!

Zur Erlangung des akademischen Grades eines
Doktors der Wirtschaftswissenschaften
(Dr. rer. pol.)
an der Fakultät der Wirtschaftswissenschaften
der Universität Fridericiana zu Karlsruhe

vorgelegte
DISSERTATION
von
Dieter Fensel

Referent: Prof. Dr. Rudi Studer
Koreferenten: Prof. Dr. Detlef Seese, Prof. Dr. Bob Wielinga

Preface

...

Cub pilot: “Indeed? Why, what could you want over here in the bend, then? Did you ever know of a boat following a bend upstream at this stage of the river?”

Apprentice: “No, sir - and ***I*** wasn’t trying to follow it. I was getting away from a bluff reef.”

Cub pilot: “No, it wasn’t a bluff reef; there isn’t one within three miles of where you were.”

Apprentice: “But I saw it. It was as bluff as the one yonder.”

Cub pilot: “Just about. Run over it!”

Apprentice: “Do you give it as an order?”

Cub pilot: “Yes. Run over it.”

Apprentice: “If I don’t, I wish I may die.”

Cub pilot: “All right; I am taking the responsibility.”

I was just as anxious to kill the boat, now, as I had been to save her before. I impressed my orders upon my memory, to be used at the inquest, and made a straight break for the reef. As it disappeared under our bows I held my breath: but we slid over it like oil.

Cub pilot: “Now don’t you see the difference? It wasn’t anything but a ***wind*** reef. The wind does that.”

Apprentice: “So I see. But it is exactly like a bluff reef. How am I ever going to tell them apart?”

Cub pilot: “I can´t tell you. It is an instinct. By and by you will just naturally know one from the other, but you never will be able to explain why or how you know them apart.”

It turned out to be true.[1]

1. Mark Twain, Life on the Mississippi, 1883.

Contents

Abstract

This thesis presents the "Knowledge Acquisition and Representation Language" (KARL), which is a formal and operational knowledge specification language. KARL provides language primitives to represent knowledge according to the layers of a KADS-oriented model of expertise. The main features of KARL are:

- It providess ***epistemologically*** adequate modelling primitives allowing knowledge specifications at the so-called "knowledge level".
- It is a ***formal*** knowledge specification language, i.e. it has a declarative semantics.
- It is an ***operational*** knowledge specification language which allows prototyping.

The main object of KARL is to fill or to narrow the gap between the symbol level and the knowledge level by defining a formal language that allows modelling at the knowledge level:

- KARL is related to the symbol level because it is a language with a formal and operational semantics.
- KARL is related to the knowledge level because it abstracts from all implementational details which are concerned with the realization by a specific computer program. KARL provides the epistemological primitives which have been identified to be necessary for the description of knowledge at the knowledge level.

KARL consists of the two languages L-KARL and P-KARL. L-KARL is used to model static knowledge, P-KARL is used to model procedural knowledge (i.e., knowledge about the control flow of a problem-solving method). It is shown how these languages and their interaction are used to model the domain layer, the inference layer, and the task layer of a model of expertise. A graphical representation for most language primitives of KARL is introduced. The usefulness of KARL is demonstrated by modelling the board-game method.

Besides KARL, some other languages have been developed during the last few years which aim to formalize or operationalize KADS models of expertise. Besides all technical details, the main difference between KARL and the other languages is the fact that KARL aims to be a ***formal and operational*** knowledge specification language. The object here is not only operationalization in principle, i.e. the possibility of using a kind of theorem-proving, but also an efficient evaluation to allow prototyping.

Acknowledgements

I thank Jeffrey Butler and Gabi Rudnick who translated my private version of “English” into common English and Cornelia Riedel who “played” the KARL interpreter for the trace in chapter three.

I thank my colleagues Dieter Landes, Susanne Neubert, and especially Jürgen Angele for years of fruitful cooperation, and Rudi Studer for guiding and improving my work. Jürgen Angele was as much engaged in the development of KARL as I was. Therefore, KARL would not look like KARL without him. In addition, Rudi and Jürgen did a hard work in criticising drafts of this thesis.

I thank Henrik Eriksson for the fruitful discussions about chronological backtracking and the board-game method. Without these discussions the specification of these methods in KARL as done in chapter five would not have been possible. I was always impressed with his ability to write and read KARL code fluently. The comparison of KARL with other knowledge specification languages was done in cooperation with Frank van Harmelen. I would like to thank him not only for his cooperation in our comparison project, which was a kind of Sisyphus work, but also for his work on $(ML)^2$, which strongly influenced my work on KARL.

Last but not least I thank Dieter Wolf who taught me scientific thinking.

Introduction

I.1 The General Aim of KARL

The "***Knowledge Acquisition and Representation Language (KARL)***" (cf. [FAL91], [AFS93])[1], which is introduced in this thesis is a means of developing knowledge-based systems in the context of the MIKE approach (Model-based and Incremental Knowledge Engineering). MIKE (cf. [AFL93]) defines an engineering framework for eliciting, interpreting, formalizing, and implementing knowledge in order to build expert systems. An expert system is a computer program which solves a task requiring a high amount of knowledge and intellectual capability when a human solves it, ***and*** this knowledge is necessary for solving the task, i.e. the problem cannot be efficiently solved by a simple complete search through the variety of possible solutions.

The process which elicits, interprets, and formalizes knowledge in order to build a knowledge-based system is called ***knowledge acquisition***. Originally, knowledge acquisition was regarded mainly as a knowledge transfer process, "... transfer and transformation of problem-solving expertise from a knowledge source to a program is the heart of the expert-system development process."[HWL83] Knowledge was viewed as an already given and fixed entity which should be elicited by the knowledge engineer or, even better, by an automated knowledge acquisition tool. The metaphor "knowledge acquisition ***bottleneck***" was soon coined for describing some unexpected problems. The problems (cf. [Ber87]) arising in the course of knowledge acquisition are well known:

- There is an insufficient understanding of the experts´ cognitive abilities to solve problems in their field.
- Human expertise is based on tacit knowledge and skill.
- Knowledge is often difficult to formalize and, to a high degree, context-sensitive.

1. In addition, KARL was submitted to a workshop at the AAAI´91 (cf. [AFL91b]) and to the KAW´91 (cf. [AFL91c]).

- Experts rationalize their behavior instead of giving their real reasons.
- Experts are experts in solving problems, not in explaining their solutions.

All these problems are indications of a significant feature of knowledge acquisition. Knowledge acquisition is not a process for transferring given knowledge into a running computer program, but a process of knowledge creation. A model of expertise is created during knowledge acquisition, which differs from the cognitive foundations of expertise as seen in the expert`s ability to solve problems (cf. [Mor87a]). Each modelling process, and thus also knowledge acquisition, has some undesirable features: it is iterative, infinite, approximative, and faulty.

An early answer to this cyclic character of knowledge acquisition was rapid prototyping as a process model. ***Rapid Prototyping*** means immediately implementing the acquired knowledge in a computer program. The running system may then be used to evaluate the gathered knowledge. This method for constructing a system reflects the above mentioned cyclic nature of the modelling process. The running system provides immediate feedback to the knowledge acquisition process for revision or completion. For some time this method was therefore applied, in most cases, to construct an expert system. Nevertheless, this method has some well-known disadvantages (cf. [AFS90]):

- The knowledge engineer who implements the system has to perform many tasks at the same time. He has to analyze the given information, as well as to design, implement, and evaluate the system.
- Different knowledge aspects have to be considered and implemented simultaneously and are therefore mixed.
- The running system is the only documentation of expertise.
- The view of knowledge is determined by the chosen implementation language, and implementational details become mixed with aspects of the conceptual model.

The problems of the rapid-prototyping approach arise because the running computer program is the only means of representing the created model of expertise. To overcome these kinds of problems, [New82] distinguishes between the ***symbol level*** and the ***knowledge level***. On the knowledge level, the model of expertise (this term will be defined in the following chapters) is described in the general terms of goals, operations, and knowledge; these terms describe the relationship of goals and operations independently of a realization by an implemented program. The knowledge level can be used to describe the problem-solving method applied by an expert or expert system, such as, for instance, heuristic classification (cf. [Cla85])[1].

As stated by [Cla89] and [Cla92b], this knowledge level description of the expertise is done by an observer, i.e. the knowledge engineer, and should not be confused with the cognitive basis of human expertise. A knowledge level description of expertise is gained by observing and ***interpreting*** the expertise of a human expert. This again emphasizes the point that knowledge acquisition is a process of model creation rather than the elicitation and transferring of fixed knowledge only (cf. [Fen92]).

The differentiation between the knowledge level and the symbol level leads to an analogous

1. This conceptual description of ***reusable*** problem-solving methods (cf. [BWS87]) independently of their realization by program code seems to be one of the major consequences of this distinction.

distinction between analysis and design for the process of developing a knowledge-based system. Speaking in terms of software engineering, a knowledge level description is the specification of the system independently of its design and implementation.[1] A life cycle-oriented development process[2], with different phases and defined milestones, is then possible. A model of expertise has to be created in the knowledge acquisition phase. In the design phase and implementation phase, additional issues are considered for constructing a useful system.

Some advantages of life cycle-oriented development methods which are dealt with in the discussion in the area of software engineering are:

- Each of the phases of the life cycle reflects a different aspect within the entire process. Thus in each stage only the considered aspect has to be analyzed and planned.
- In contrast to rapid prototyping, the application of the life-cycle paradigm allows a well-structured development process and, likewise, a well-structured final result.

If this paradigm is applied to the construction of expert systems, this process is structured in a manner which justifies using the term knowledge ***engineering***. Apart from their advantages, life cycle-oriented process models still suffer from a significant problem. They require fully elaborated documentations of the results of every phase. These documentations are expected to be correct, complete, consistent, and unequivocal. But the documentation of the first phase, the model of expertise, is normally informal or semiformal. It is neither possible to analyze it automatically, nor to execute it. Therefore this important document generally contains a lot of inconsistencies and ambiguities.

Three different ways of compensating these shortcomings are possible without losing the advantages of the life-cycle oriented development process:

- The shortcomings of life cycle-oriented development processes discussed above arise because these process models do not consider the cyclic character of the system development process. More adequate process models therefore try to integrate the different phases and steps into one cyclic and reversible process model. Examples in the area of software engineering are the ***spiral model*** (cf. [Boe86], [Boe88]) or ***evolutionary prototyping*** (cf. [Flo84], [FRS89]).
- Formal specification languages are developed which improve the result of the specification phase. In [AHS93] some of the advantages of a formal description of the model of expertise are summarized:
    - A formal description reduces the vagueness and ambiguity of natural-language descriptions.
    - A formal description helps to obtain a clearer understanding of the individual problem-solving steps and the entire problem-solving methods, thus supporting potential reuse.
    - A formal description can be mapped to an operational description to permit prototyping, which supports knowledge evaluation.

---

1. For the differences between the purpose of a specification in traditional software engineering and knowledge engineering see [AFS90].
2. The stagewise model (cf. [Ben56]) and the waterfall model (cf. [Roy70]) are early examples.

- A formalized specification allows symbolic execution for evaluating the model of expertise and makes checks for consistency and completeness possible.

- Operational specification languages are developed to improve the result of the specification phase. Prototyping can be used to evaluate the specified knowledge. In [Flo84] this kind of prototyping is called ***explorative*** and ***experimental prototyping***. Prototyping can be applied to detect errors or to assess the modelled solution, i.e. the chosen problem-solving method.[1]

The development of a process model and of means for the knowledge engineering process which compensate the shortcomings discussed above are the basis of the MIKE approach (Model Based and Incremental Knowledge Engineering) (cf. [AFL93]). It aims at integrating the advantages of life-cycle models, of prototyping and formal specification techniques into a coherent framework for the knowledge engineering process. The main characteristics of MIKE are

- a spiral model as process model for the knowledge-engineering process (cf. [AFS90], [AFS90], [NeS92]),
- a Hypermodel which is embedded in a hypermedia environment (cf. [HoN92]) and which defines a semiformal knowledge representation for the early knowledge acquisition phase, and
- a formalization and operationalization of models of expertise in KARL.

This thesis presents the "Knowledge Acquisition and Representation Language" (KARL) (cf. [FAL91], [AFS93]), which is a formal and operational knowledge specification language. KARL provides language primitives to represent knowledge according to the layers of a KADS-I-oriented ***model of expertise*** (cf. [BrW89], [WSB92])[2]. The main features of KARL are:

- It provides ***epistemologically*** adequate modelling primitives allowing knowledge specifications at the so-called "knowledge level". It distinguishes three different types of knowledge (i.e., domain knowledge, inference knowledge, and control knowledge) and defines different language primitives for them.[3]
- It is a ***formal*** knowledge specification language: KARL has a declarative, i.e. model-theoretic, semantics.[4]
- It is an ***operational*** knowledge specification language, i.e. its semantics can be operationalized and implemented, which allows prototyping.[5]

Additional features are:

- Complex specifications, as required for real-life systems, easily tend to be difficult to oversee. Therefore, ***hierarchical refinement*** and ***structuring primitives*** are given.

1. Cf. [Ang93] for a discussion of prototyping with KARL.
2. KADS project (ESPRIT-I P1098).
3. "The system specification must be a cognitive model rather than a design or implementational model." [BaG79], p. 59
4. "... a specification is a description of ***what*** is desired, rather than ***how*** it is to be realized (implemented)." [BaG79], p. 58
5. "A specification must be operational." [BaG79], p. 59

- Most of KARL´s modelling primitives have a ***graphic*** representation to allow KARL specifications to be used as a medium of communication.

KARL is not designed to be a language for describing the final knowledge-based system in the design phase or implementation phase. One central idea of KADS is the separation of analysis and implementation. Consider software engineering as an analogy: a specification language was created to allow executable and formal specifications. KARL utilizes results of software engineering and information system developments.[1] Both disciplines have a long tradition in developing and applying formal and executable specification languages (cf. [Alf90], [Fea87], [Kun89], [LNO89], [Say90], [Zav91]). The domain layer of KARL uses ***semantic*** and ***object-oriented data models*** (cf. [Bee90], [BrR84], [Che76]) to represent static knowledge. The inference layer applies concepts of ***Structured Analysis*** (cf. [Ros77], [You89]), and the task layer language is influenced by languages like ***RSL*** (cf. [Alf90]).[2]

I.2 A Model of Expertise in KARL

The KADS-I project aims at the development of a comprehensive, commercially viable methodology for knowledge-based system construction. The entire construction process was divided into several subactivities, and documents as results of these activities were defined. In the following, only one aspect of the entire KADS methodology will be discussed. The ***model of expertise*** according to the KADS-I project (cf. [BrW89], [WSB92]) is the result of the knowledge acquisition activity and specifies the problem-solving expertise required to perform the problem-solving tasks assigned to the system. A model of expertise according to the KADS-I project distinguishes different kinds of knowledge and provides different epistemological primitives for each of these types of knowledge.

The ***domain layer*** contains domain-specific knowledge about concepts, their features and their relationships. At this layer, the objects, their features, and the terminology of the application domain are described. KARL clearly separates knowledge and data. The knowledge part of a domain layer contains domain knowledge and therefore cannot change during a problem-solving process of a case, otherwise the knowledge acquisition step or a kind of machine learning would be modelled. All case-specific data must be read from the input-data part and written on the output-data part separately from the other domain layer. Intermediate results of a problem-solving step must all be represented at the inference layer. Therefore, inference actions can only share common data if they are connected by common roles (i.e. stores). This hinders the domain layer from being a global variable with read access and write access.

KARL provides a sub-language ***Logical-KARL (L-KARL)*** to describe the case-specific data and the knowledge at the domain layer.

- L-KARL distinguishes between objects which model real-world entities and attributes which model features of such entities as in object-oriented data bases (cf. [Bee90]).[3] It provides predicates, classes, class hierarchies, signature terms, and multiple-attribute

1. Cf. [AFL91d]
2. See chapter 6.2 of the thesis.
3. Cf. [Kim90] for an introduction to object orientation.

inheritance for modelling terminological domain knowledge (cf. F-logic [KiL89], [KLW90]). Attributes can be functional, i.e. the same object cannot have two different values for an attribute, or set-valued, i.e. the value of an attribute could be a set of values. The derivation of new object denotations can be expressed by functions. O-logic (cf. [KiW89], [KiW93], [Mai86]) and F-logic ([KiL89], [KLW90]) were used here as models for the integration of object orientation into a well-defined declarative framework.

- L-KARL provides Horn logic extended by stratified negation (cf. [Prz88]) to express intensional descriptions, i.e. sufficient conditions, and factual knowledge.
- A richer logical language is provided for formulating constraints, which are necessary conditions for concepts and relationships.
- The model-theoretical semantics of the domain layer is the perfect Herbrand model (cf. [Prz88], [KLW90]).

The ***inference layer*** contains knowledge about the used problem-solving method. This layer indicates which inferences are necessary within the problem-solving method. This knowledge describes the application of the domain knowledge for making inferences. It describes the logical inferences and their dependencies. No causal dependencies or causal ordering of these logical implications are given. The inference layer consists of two types of elements: ***roles*** and ***inference actions***.

- Three types of roles are given by KARL. ***Views*** define an upward view from the inference layer on the domain layer using the logical language described above. The result of a view need not correspond to a predefined domain concept or to a predefined domain relationship. Several domain expressions can be combined to form a new expression by carrying out a view definition using L-KARL. A view defines a read-access to the knowledge and input-data part of a domain layer. ***Terminators*** define a downward view from the inference layer on the domain layer using the logical language described above. The results of a problem-solving process at the inference layer are rephrased in domain-specific terms using L-KARL. A terminator defines a write-access to the output-data part of a domain layer. ***Stores*** provide the input for or collect the output from inference actions. Whereas views and terminators are used to link a domain layer with a generic inference layer, stores are used to model the dataflow dependencies between inference actions.
- Inferences are described by means of inference actions. An ***elementary inference action*** is described declaratively by a set of logical clauses using the logical language L-KARL.
- A store can be a conclusion of one and a premise of another inference action. An ***inference structure*** is a network which shows these dependencies between stores, terminators, views, and inference actions.
- ***Levelled inference layer:*** A subset of an inference structure can be arranged to form one ***composed inference action***, vice versa an inference action can be refined to a network of inference actions and roles. These composed inference actions can be used as elements to specify inference steps. The inference processes can therefore be described at several levels of refinement. The main advantages of a levelled inference layer are (cf. [You89]):
  - The modelling of complex systems within one single inference structure would result in an unreadable inference layer.

- It is possible to define several views of a system or a problem-solving method. Every view shows the system or the method at a different level of refinement.
- The top levels of inference structures which are developed bottom-up can be reused for similar tasks.
- If a system requires the cooperation of several predefined problem-solving methods, any of these methods can be represented by a composed inference action in one inference structure. The representation of the interactions of the methods is not confused with their internal structure.

The semantics of a composed inference action is defined by its basic elements and the control flow between them as defined at the task layer. The refinement of an inference action stops if its behavior can be described declaratively without any control knowledge concerning the order of activities. This kind of control knowledge is described at the task layer by determining the sequences of inference actions. If a step represented by an inference action contains control knowledge which should be represented at the model of expertise, the inference action has to be broken off. If an inference step represented by a inference action does not contain any control knowledge there is no need to refine it.

The ***task layer*** contains knowledge about the control flow of a problem-solving method for solving a specific task. This layer specifies sequence, iteration, and alternative of inferences. The logical language ***Procedural-KARL (P-KARL)*** is provided for this purpose. It is a variant of dynamic logic (cf. [Har79], [Har84], [Koz90]) and therefore allows the declarative description of a control flow. Extended by additional syntactical sugar it can be used in a similar manner as procedural languages like Pascal or Modula-2. The elementary steps at the task layer correspond to the elementary inference actions. Stores are modelled by variables. Composed inference actions are represented by subtasks. These expressions can be arranged by sequence, loop, and alternative in order to express the control flow.

The ***strategic layer*** contains knowledge about the relationships of tasks and problems, i.e. it is used to dynamically arrange the appropriate steps of a problem-solving process in order to solve a specific problem. Owing to the fact that there exists little agreement in the literature about the definite purpose of this layer and to the epistemological difference of the knowledge represented by it and the knowledge which is represented at the task layer, this layer is not discussed further on. The vagueness of the description of this layer corresponds to the lack of identified standardized primitives to express this knowledge. It was beyond the scope of this thesis to develop new primitives to express different types of knowledge; the object here was rather to define a formally defined meaning for the existing ones. Even the formal definitions of already given primitives required further refinements and modifications of them.

I.3 Graphical Representation

As a rule, formal descriptions like first-order logic are difficult to understand. Formal specifications are therefore not an adequate tool for communicating with experts or system users (cf. [Fet88], [FrD89], [GeM86], [MaM85], [Nau82]). KARL provides graphical representations of most modelling primitives to improve their understandability. Every graphical symbol has a defined meaning given by the semantics of the corresponding language primitive:

- Enhanced-Entity-Relationship (EER) diagrams (cf. [ElN89]) were the models for the graphical primitives at the domain layer,
- levelled dataflow diagrams (cf. [You89]) were the models for the graphical primitives at the inference layer, and
- program-flow diagrams (cf. [Din66001]) were the models for the graphical primitives at the task layer.

Specifications of real-world systems become unintelligible because of their complexity. Therefore all three graphical representations include hierarchical refinement to allow the representation of the system at different levels of refinement.

I.4 The Formal Level

The logical language L-KARL used to describe the domain layer, the inference layer, and their connection has a model-theoretical semantics (cf. [Llo87]) similar to other first-order logic languages. As allow negative information, i.e. negation of premises of rules is allowed, L-KARL applies the closed-world assumption and uses the minimal, i.e. perfect, Herbrand model as semantics (cf. [Prz88], [KLW90]) of a set of clauses. The constraints check this model for correctness.

The procedural knowledge is represented by P-KARL, which also has a model-theoretic semantics. It is a variant of Quantified Dynamic logic (cf. [Har79], [Har84], [Koz90]), which can be used to declaratively describe procedural programs.

The perfect models of L-KARL are used to define an interpretation for a P-KARL language, i.e. the perfect Herbrand model of a set of clauses and is applied to interpret a function symbol used in value assignments in P-KARL.

I.5 The Implementational Level

An earlier version of KARL is mapped to PROLOG to make it executable. The mapping is also implemented in PROLOG (cf. [Böh92]). The running prototype has been proved to be very useful for the evaluation of the model of expertise. This evaluation process is additionally supported by a debugger which allows to trace the single inference steps and to examine the contents of the stores, thus making the problem-solving process with all its intermediate data transparent for the knowledge engineer. For the creation of the model, a hypertext-based tool has been implemented which allows to structure verbal protocols in a so-called Hypermodel. These structured informal or semiformal descriptions serve as input for the formalization process (cf. [HoN92]). Additionally, the corresponding parts of the formal model and the structured informal Hypermodel are linked explicitly and thus provide documentation facilities for the formal model. Because of the additional features of the new KARL version and disadvantages of the evaluation strategy of PROLOG, a new interpreter for KARL is under development which should also be more efficient than the PROLOG-based interpreter (cf.

[Ang93]).

I.6 Case Studies

KARL has already been applied to solve the following problems:

- ***Assignment***: The “Sisyphus problem” (cf. [Lin92c], [Lin92d]) has been brought up recently; it allows a comparison of different modelling approaches and languages. The problem in Sisyphus consists of assigning a set of employees to appropriate office rooms. When doing this, certain requirements have to be taken into account, for instance, certain persons have preferences for certain rooms, rooms may be shared by two persons and, if so, these persons should get along with each other, etc. ([AFL91a], [AFL92b]). In [LFA93], KARL is applied for specifying a solution for a simple design task, like scheduling a set of activities to a set of time periods with respect to some given requirements. A KARL model for a problem in the area of air pollution control is described in [LHS92]. This application is concerned with designing an optimal emission control procedure, i.e. a combination of individual emission control technologies, for a given plant. Difficulties arise because certain techniques affect more than one class of pollutants, and the application of some techniques excludes others from being applied. Furthermore, to permit the comparison to alternative emission control measures, a flexible optimality criterion must be introduced which is based on aspects such as the total annual cost, the technical maturity of the considered technologies, the extent to which pollutants are reduced, etc. depending on the individual preferences of the plant’s operators.
- ***Selection***: The selection of scheduling algorithms for project management is also modelled in KARL (cf. [KFG92]). Starting from a description of the context in which the scheduling algorithm should be applied, the first step of the problem-solving method concentrates on selecting a set of possible algorithms which match the given problem structure. These algorithms are ordered according to certain preference criteria, then the best are chosen and, finally, their assumptions are tested to check if they are really applicable in the given situation. If no suitable algorithm can be found, it is possible to point out discrepancies between the initial problem description and descriptions of available algorithms, thus allowing a user to modify his/her problem description such that one of the available models can be used.
- ***Cover and differentiate***: In addition to these applications which result in complete models of expertise, i.e. which also encompass a domain layer, KARL has also been used in [Ang92] to remodel the generic problem-solving method cover and differentiate” (cf. [Mar88]). This problem-solving method is suitable for problems in which a solution has to be identified from a given set of potential solutions, like diagnosis problems. Cover and differentiate first identifies hypotheses which explain the observed symptoms, and then uses observables to differentiate hypotheses. This is repeated several times since the remaining hypotheses may be explained by other hypotheses, which may again be differentiated by observables. Thus, hypotheses are arranged in a causal network, which is traversed by cover and differentiate in a breadth-first manner, with bad paths being ruled out by the use of differentiating knowledge.

- ***Board-game method***: In this thesis the so-called “board-game method” (cf. [EST92]) is modelled in KARL. 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. The method can be applied to assignment problems like the Sisyphus problem (cf. [Lin92d]) or to games like “Tower of Hanoi”.
- In [FAL93] it is described how KARL can be used to formalize and operationalize software specifications using ***Structured Analysis techniques*** (cf. [You89]). The ability to specify common software engineering techniques in KARL allows the integration of specifications of knowledge-based components into the specification of larger systems consisting of conventional components.
- Further applications are the German Umweltverträglichkeitsprüfung for planning traffic lines (cf. [Güh92]), a decision support system for companies related to ecological requirements (cf. [Bic92]), and the design of insurance packages (cf. [Ohl92]).

I.7 Overview

The overview given below provides a concise description of the main characteristics of KARL.

Overview of KARL

(1). Domain layer	- Classes, predicates, “is-a” and “is-element” relationships, functional and set-valued attributes, and multiple attribute inheritance. - Extensional and intensional descriptions of concepts and relationships by clauses. - Constraints. Language: object-oriented Horn logic with stratified negation (L-KARL, which is restricted and modified F-logic).
(2). Inference layer	- Elementary inference actions; - composed inference actions (levelled inference layer); - L-KARL to describe inference actions; - roles (stores, terminators, and views).
(3). Task layer	- Sequence, branch, loop; - hierarchical refinement (subtasks). Language: P-KARL (restricted version of dynamic logic).
Mapping: domain layer to inference layer	Arbitrary view definitions by object-oriented Horn logic extended by stratified negation.

Overview of KARL

Mapping: domain layer and inference layer to task layer	The languages of the domain layer and of the inference layer define a semantic structure for the language of the task layer.
Special features	- Applies object-orientation; - allows hierarchical structuring of complex models of expertise; - has a formal semantics; - is an operationalization and formalization language.

I.8 Content of the Thesis

Chapter one introduces the basic idea of KARL. The main object of KARL is to fill or to narrow the gap between the so-called symbol level and knowledge level (cf. [New82]) by defining a formal language allowing modelling at the so-called knowledge level. Chapter two defines the syntax and semantics of KARL. KARL consists of the two languages L-KARL and P-KARL. L-KARL is used to model static knowledge, P-KARL is used to model procedural knowledge, i.e. knowledge about the control flow of a problem-solving method. Chapter three shows how these languages and their interaction are used to model the domain layer, the inference layer, and the task layer of a model of expertise. Chapter four introduces graphical representations for most language primitives of KARL. The usefulness of KARL is demonstrated in chapter five where it is used to model the ***board-game method***.

A lot of work has been achieved in recent years in developing operational and formal knowledge specification languages, like FORKADS (cf. [Wet92]), K_{BS}SF (cf. [JoS92]), $(ML)^2$ (cf. [HaB92]), Model-K (cf. [KaV92]), MoMo ([WVL92]), OMOS (cf. [Lin92a]), and QIL ([ARS92]), for example. Chapter six compares KARL with these languages (cf. [FeS92], [FeH93]) and adds a short comparison with languages which originate in the software engineering domain (cf. [FAL93]). A comparison of KARL and the language INCOME, from the domain of information system development, can be found in [AFL92a].

This thesis, only discusses the formal and epistemological aspects of KARL, i.e. its operationalization and the operational semantics of KARL are not considered here. The operationalization of KARL and its use for prototyping is discussed in [Ang93].

I.9 What Distinguishes KARL from all other Languages?

Besides KARL, other languages have been developed during the last few years with the object of formalizing or operationalizing KADS models of expertise. Apart from all technical details the main difference between KARL and the other languages is in KARL´s claim to be a ***formal and operational*** knowledge specification language. The aim here is not only operationalization in principle, i.e. the possibility of using a kind of theorem-proving, but also an efficient evaluation to allow prototyping. Languages like Model-K, MoMo, and OMOS aim at operationalization only. Languages like $K_{BS}SF$, $(ML)^2$, and QIL are intended as formalization languages. For $(ML)^2$ and QIL executability in principle is discussed by the use of theorem-provers, but it is not a main goal which influenced the design of the language. Whereas $(ML)^2$ gives first-order logic, KARL restricts the logical language to Horn logic in order to allow efficient evaluation. Only FORKADS is congenial to KARL because it is also intended to be a a formal and operational language. However, FORKADS does not have a declarative semantics and therefore fails to meet an essential requirement for a formalization language.

As KARL tries to integrate the formalization aspects and operationalization aspects into one language, it has been interesting to observe if and to what extent these different requirements are either contradictionary or complementary (cf. subsection C.2.2 of the conclusion).

- ***Complementary***: Stratified Horn logic with its perfect model semantics has been chosen as logical language for the domain layer and the inference layer because this is a quite natural choice for further operationalization. Therefore, the same language is provided for formalization and operationalization, and the declarative and operational semantics are identical in principle. The use of simplified dynamic logic was inspired by $(ML)^2$ and allows the declarative and operational description of procedural knowledge.
- ***Contradictionary***: The main difference between operational and formal KARL is the fact that the operationalization of KARL requires finite extensions of predicates, i.e. finite sets of P-terms and F-terms which are true in the perfect Herbrand model of a set of clauses and definitions. This requirement does not hold for KARL when considered "only" as a formal specification language. This restriction does not change the semantics of the language but restricts its expressive power at the domain layer and the inference layer. Further restrictions concerning well-typing and equalities, which modify the semantics, are defined in [Ang93] to improve the efficiency of the implemented evaluation algorithms. They are not necessary in principle, but they allow a more efficient evaluation of KARL and improve prototyping with it.

Chapter 1

Model-based and Incremental Knowledge Acquisition

This chapter will first discuss the characteristics of knowledge acquisition and show how they motivated the development of the “Knowledge Acquisition and Representation Language” (KARL) (cf. [FAL91], [AFS93]). Then subtasks of knowledge acquisition will be discussed, and it will be pointed out how KARL is related to some of them.

1.1 The Reconciliation of Symbol Level and Knowledge Level

This chapter is to recall an “old” discussion, which started a decade ago, when Newell introduced the so-called ***knowledge level*** (cf. [New82]) in order to open a new view on knowledge-based systems. The discussion about knowledge was led without referring to a special representation and inferencing mechanism used to implement the knowledge-based system. Also, in other respects there were some significant weaknesses in Newell`s knowledge-level approach in its original version. It was the development of generic problem-solving methods, and the distinction of different types of knowledge and primitives to express them, which actually provided this level with a semantics besides the very abstract and vague characterization by Newell.

In addition, this new view on knowledge allowed a new view on knowledge acquisition, i.e. to recognize knowledge acquisition as a modelling activity.[1] There is no existing frame or production rule in the mind of an expert which must be mined like a diamond out of the earth, but the model of expertise is the result of an investigation and interpretation of his ability to solve problems. The model of expertise is an artificial product of a modelling activity.

1. A new point of view on a product implies a new point of view on the process which creates it.

However, the informal descriptions of generic problem-solving methods and domain-specific knowledge which constitute such models of expertise suffer from significant disadvantages. They are neither operational, which would allow their evaluation by prototyping nor have they a formal semantics, which would allow checks for consistency, completeness, and correctness.

This was the reason for developing formal and operational knowledge specification languages like (ML)^2 ([HaB92]) or KARL, which include modelling primitives recognized to be suitable for describing models of expertise or knowledge at the knowledge level (cf. [WSB92]). The knowledge level is used to describe the problem-solving method and domain-specific knowledge of agents which should be able to solve the problems in specified tasks and domains. It abstracts from implementational details concerned with an efficient implementation by means of a specific programming language. With this kind of languages the old contradiction of symbol level and knowledge level has been overcome. They are formal and operational and therefore related to the symbol level. They are related to the so-called knowledge level because they define modelling primitives which are needed at this level to describe models of expertise. In analogy to Brachman (cf. [Bra85]) this missing link between both levels that is defined by these languages is referred to as the ***epistemological level***.

1.1.1 The Separation of Symbol Level and Knowledge Level

> "***The Physical Symbol System Hypothesis***. A physical symbol system has the necessary and sufficient means for general intelligent actions. ... This is an empirical hypothesis." [NeS76], p. 116

[NeS76] introduces the physical symbol system hypothesis; it is refined and extended in [New80]. A physical symbol system consists of a memory for storing expressions, of a set of operators which can be used to manipulate expressions, a control, an input, and an output (cf. [New80], p. 142). It has the ability

- to ***designate*** real-world objects by symbols or expressions
- to ***interpret*** given symbols or expressions, to execute the process which is denoted by a symbol, for example.

This physical symbol system is universal in the sense that it lends itself to the simulation of Turing machines and can therefore be used to compute every computational function (cf. [HoU79]).[1]

The description of intelligence by a symbol system is the paradigm which underlies the general problem-solving approach. A declarative logical description of a problem and of the intended solution (in first-order logic) is to be used to compute a solution by applying a general search procedure. A theorem-prover can be applied for solving arbitrary tasks in arbitrary domains by a general search strategy. As the number of derivable facts increases exponentially, this strategy can perform a task only in an exponentially increasing period of time.

This dilemma gave rise to the saying that "knowledge is the power" [Fei80], which means that only a high amount of task-specific and domain-specific knowledge and its appropriate use

1. The implicit assumption of Newell and Simon is that this is sufficient for intelligent behaviour.

allow to tackle problems for the solution of which humans must rely on their intelligence. But the representation formalisms which were available to represent this knowledge, like the production rule system or semantical networks, caused several significant problems:

- ***Ontologically different kinds of knowledge were represented indiscriminately***: Domain-specific knowledge was represented in the same manner as common-sense knowledge or knowledge about the dialogue strategy of an expert system (cf. [Cla83], [Cla86]). The following two MYCIN rules represent different kinds of knowledge. The first rule (1) represents domain-specific knowledge, whereas the second (2) uses common-sense knowledge to control the dialogue strategy (in the present example: a person which is younger than eighteen should not be asked whether he is an alcoholic):
    - "(1) If the patient is less than seven years old, then remove tetracycline from the list of drugs under consideration." [Cla86], p. 45.
    - (2) If the age of the patient is greater than 17 and the patient is an alcoholic, then Dipococcus might be causing the infection." [Cla86], p. 43.[1]
- ***Epistemologically different types of knowledge were represented indiscriminately***: Production rules were designed as a declarative representation formalism. Experiences in XCON (cf. [SBJ87], [BaS89]) have shown that great problems arise if the control flow is implicitly specified, e.g. in production rules only. The experts have control knowledge, i.e. knowledge about a meaningful ordering of the different steps of a problem-solving process, and this knowledge is necessary for an efficient solution. Therefore, they represented it implicitly by encoding an implicit sequence of the rules. A similar experience was made with PROLOG and KL-ONE-like representation formalisms. "Just as the rule notation was >>abused<< in MYCIN by ordering rule clauses to encode hierarchical and procedural knowledge, users of KL-ONE implicitly encode knowledge in structural properties of concept hierarchies, relying on the effect of the interpreter to make correct inferences.“ [Cla85], p. 336.
- ***Implicit representations of heuristics, strategic knowledge, and problem-solving methods***: [Cla85] analyzed several expert systems for diagnosis and remodelled the so-called problem-solving method of heuristic classification (cf. below), which was implicitly used by all of these expert systems without an explicit representation of it.
- ***Uniform representation of different kinds of control knowledge***: Control knowledge of a problem-solving method is mixed with control knowledge of a specific representation formalism which is used to implement it. "The description of the diagnostic strategy of NEOMYCIN in [Cla85] is a clear example of knowledge-level control. This type of knowledge concerns task decomposition of and ordering between knowledgeable agents and possibly also meta-knowledge about agents. Symbol level control is concerned with the control issues that arise when a particular representation or AI technique is selected to realise a problem solving agent.“ [Sch92a], p. 12

It was Newell`s merit (cf. [New82]) to formulate the conceptual framework to overcome this shortcomings. He introduced the so-called knowledge level.

1. Because of the evaluation strategy the evaluation of the rule premises stops after a premise has failed.

> "***The Knowledge Level Hypothesis***. There exists a distinct computer systems level, lying immediately above the symbol level, which is characterized by knowledge as the medium and the principle of rationality as the law of behavior." [New82], p. 99

On the one hand, this was a great innovation. Instead of discussing knowledge in terms of its symbolic representations and inference machines which are applied to work with these knowledge representations, now knowledge itself became a subject of the discussion. "Representation exists at the symbol level ... Knowledge serves as the specification of what a symbol structure should be able to do." [New82], p. 100

On the other hand, Newell was unable to characterize this level in more than very vague and abstract terms. He is aware of this deficit, which is reflected in his pointing out four striking features:

- The components of a system at the knowledge level are: actions, a body of knowledge, and a set of goals. "There are no laws of composition for building a knowledge level system out of these components. ... This complete absence of significant structure in the agent is the first surprise, ..." [New82], p. 101
- The behavior of an agent is determined by the so-called principle of rationality. "If an agent has knowledge that one of its actions will lead to one of its goals, then the agent will select that action. ... This direct determination of behavior by a global principle is the second surprise, running counter to the common feature at all levels that behavior is determined bottom-up through the local processing of components." [New82], p. 102
- "However, the principle is not sufficient to determine the behavior in many situations. ... This failure to determine behavior uniquely is the third surprise, running counter to the common feature at all levels that a system is a determinate machine." [New82], pp. 102ff
- "Knowledge is to be characterized entirely ***functionally***, in terms of what it does, not ***structurally***, in terms of physical objects with particular properties and relations. ... This failure to have the medium at the knowledge level be a state-like physical structure is the fourth surprise, running counter to the common feature at all levels of a passive medium." [New82], p. 105

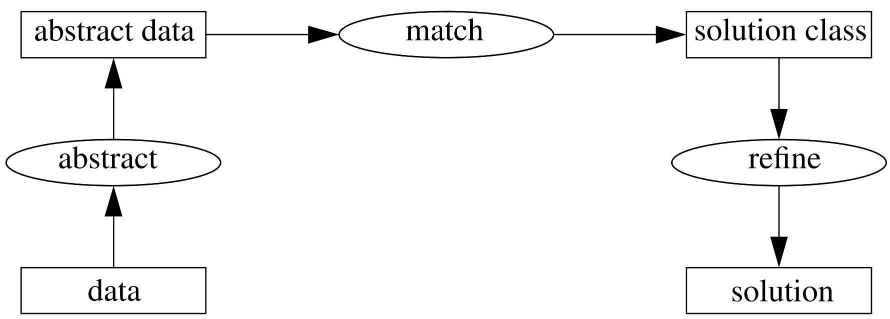


Figure 1. Inference structure of heuristic classification

It was the pioneer work of Clancey (cf. [Cla85]) which allowed a much more concrete view at the knowledge level. He compares several expert systems and extracts a common method used by these systems, which he calls ***heuristic classification***. Apart from various differences all systems have three steps in common:

- a data-abstraction, i.e., concrete values are mapped on intervals;
- a heuristic match from abstract problem descriptions to abstract solutions;
- the hierarchical refinement of these solutions until a final solution has been found.

Thus, Clancey describes a problem-solving method at the knowledge level without referring to implementational details of the used knowledge representation formalism of the different systems. In [BWS87] a whole set of different problem-solving methods is described at this level. These problem-solving methods are described by defining

- the single steps called knowledge sources or inference actions;
- the structural dependencies of these single steps by a so-called inference structure;
- the control flow, i.e. the sequence of these single steps.

Thus, knowledge becomes structured and can be described by a much more concrete law of behavior than stated by Newell. It is mainly the work of the KADS-group (cf. [BrW89], [WSB92]) which defines different types of knowledge and different primitives for expressing them:

knowledge category	**organization**	**knowledge types**
strategic	strategies	plans and meta rules
↓ *controls*		
task	tasks	goals, control terms, and task structure
↓ *applies*		
inference	inference structure	inference actions, knowledge roles, and domain view
↓ *uses*		
domain	domain theory	concept, property, and relation

Figure 2. The four-layer model of expertise (cf. [WSB92]).

- The ***domain layer*** contains domain-specific knowledge about concepts, their features, their elements, and their relationships. This knowledge is task-neutral in the sense that there are no means to control its use. At this layer there is a description of the objects and the terminology of the domain which the knowledge-based system is to use.
- The ***inference layer*** contains knowledge about the used problem-solving method. This layer indicates which inferences are necessary within the problem-solving method. This knowledge describes the application of the domain knowledge for inferencing. It describes the logical inferences and their dependencies. No causal dependencies or causal ordering of these logical implications are given.
- The ***task layer*** contains knowledge about the control flow of a problem-solving method which is to solve a specific task. This layer specifies when inferences are made.

- The ***strategic layer*** contains knowledge about the combination and selection of tasks which are to solve a problem adequately.

This new semantical and syntactical view[1] at the knowledge level allows a new perspective of one of its features already mentioned by Newell:

> “***Knowledge***. Whatever can be ***ascribed*** to an agent, such that its behavior can be computed according to the principle of rationality. ... One way of viewing the knowledge level is as the attempt to build as good a model of an agent´s behavior as possible based on information external to the agent, hence permitting distal prediction.“ [New82], p. 105, 108f

1.1.2 Knowledge Acquisition as a Modelling Process

> “Knowledge-level descriptions are attributes made by an observer (the knowledge engineer), involving his own selective interactions with the agent (the expert), his own perceptions, and his point of view; ...“ [Cla89], p. 288

The description of a computational model of the expertise even at the knowledge level differs significantly from the cognitive foundation of an expert`s ability to solve problems. His expertise is based on a large variety of experience which is compiled in a high amount of skill and implicit knowledge (cf. [DrD86]). Domain-specific heuristics rather than generic problem-solving methods are the guidance of a real-world expert. Therefore, these knowledge level descriptions are ***constructed***; they have the expert`s expertise as an empirical foundation but they abstract and modify it. They are a model, and knowledge acquisition is mainly a modelling activity.

> “Knowledge acquisition is a constructive modeling process, not simply a matter of >>expertise transfer<<.“ [FBA93], p. 9.

The main task of knowledge acquisition is not the transfer of already existing knowledge but the creation of a computational knowledge model. Knowledge acquisition is not intended to create an impartial description of the expertise of an expert. The knowledge engineer has to create a new computational model in cooperation with the expert; as a rule this model does not exist before the process. Therefore, influence of the knowledge engineer and modification of the expertise is no error which must be minimized, but it is even a key feature of the process. Explaining, structuring, and generalizing the expertise during the knowledge acquisition process extends its reliability, validity, and applicability. The learning process of the expert can improve the elicited knowledge (cf. [Fen92])[2].

Considering knowledge acquisition as a kind of theory building[3] gives insight into some of its fundamental features (cf. [Mor87a]):

---

1. One could consider the different layers and their according primitives as syntax of the knowledge level and the generic problem-solving methods as its semantics. Otherwise, this is also misleading because these different layers have their own semantics by the epistemological different knowledge types.

2. Sometimes he learns so much about his own skills that an expert system is not needed anymore.

3. [Nau85] and [Fic87] take a similar point of view for every software development activity.

- Modelling is an ***infinite process***, because every model is only an ***approximation*** (cf. Newell´s third "surprise").
- Modelling is a ***faulty process***, i.e. there is a high need for conservative and revolutionary ***revision*** (cf. [Emd91]) of already modelled parts.
- Modelling is an ***incremental process***, i.e. the knowledge engineer becomes familiar with the expertise only gradually and must use the intermediate results as ***guidelines*** for further activities. A first judgement is the foundation for the first perception and interpretation of the object of research. This interpretation creates a new understanding. This procedure must be repeated several times. If the process works well it leads to an increasing understanding. The process can be illustrated graphically by a spiral called ***hermeneutic circle***.

[Mor87a] designates her proposal for knowledge acquisition as ***sloppy modelling***. There is no complete and perfect model of the expertise in the expert`s mind, but the expertise is based on experience, vague heuristics and tacit knowledge. Therefore a model is created as a ***result*** of the knowledge acquisition process. Methods for knowledge acquisition must therefore meet the following requirements:

- They must allow iterative modelling.
- They must be suitable for detecting errors.
- They must support error corrections.

The incremental development of a ***running prototype***, its evaluation and modification meet the requirements given above. The usefulness of prototypes, i.e. operational specifications, for the evaluation of system specifications is well-known in the domain of software engineering (cf. [Flo84]). Informal or semiformal specification of software products, which correspond to informal or semiformal knowledge level descriptions of expert systems, have some well-known disadvantages (cf. [AFS90]):

- Natural-language documents have an ambiguous and vague semantics.[1]
- There is a high cognitive distance between the specification and the implementation of a system, i.e. the specification is not an appropriate means for defining the behaviour of the desired system.
- As the specification is not executable, there is little support in evaluating it.
- As the specification is normally only an informal or semiformal one, there is less support in verifying its correctness and completeness.

These disadvantages of informal knowledge level descriptions correspond to the advantages of a formal description of a model of expertise (cf. [AHS93]):[2]

1. In general, ambiguousness and vagueness of natural language are not disadvantages of natural language but an essential and necessary feature of it. Subjects with different interpretations of reality can only develop common patterns of interpretation without enforcing their absolute equality by such a means. But a computer program is no agent which interprets reality but it is a deterministic natural process which requires a detailed and precise description when it should be manageable.

2. A formal descriptions should not replace an informal description but should add a more precise and detailed level of descriptions. Informal and formal descriptions are complementary and not contradictionary.

- A formal description reduces the ambiguity and vagueness of natural language descriptions.
- A cognitive distance between the specification and the implementation is lower and the specification can be used to prove the implementation for correctness.
- A formal description can help to get a clearer understanding of individual problem-solving steps and complete problem-solving methods, thus supporting potential reuse.
- A formalized specification can be mapped on an operational specification, which allows prototyping for supporting knowledge evaluation.
- A formalized specification allows the verification of the modelled expertise, which makes checks for consistency and completeness possible.

However, using operational or formal languages for knowledge level descriptions instead of informal or semiformal languages may cause difficulties because the separation of symbol level and knowledge level may be endangered by their use. The key idea of the knowledge level description is the description of expertise which is independent of any specific symbolic representation used later on to implement a running computer program. How can this contradiction be solved?

1.1.3 The Reconciliation of Symbol Level and Knowledge Level

Brachman identifies four levels which help to compare and characterize different semantical networks (cf. [Bra85]):[1]

- ***Implementational level***: “In implementation level networks, links are merely pointers, and nodes are simply destination for links.“ [Bra85], p. 204
- ***Logical level***: “In this type of *logical* level net, nodes represent predicates and propositions. Links represent logical relationships, like AND, SUBSET, and THERE-EXISTS.“ [Bra85], p. 204
- ***Conceptual level:*** “This is, in some sense, the >>real<< semantic net - a network structure whose primitives are word-sense and case relations. As Simmons notes, in these *conceptual* level nets, links represent semantic or conceptual relationships. Networks at this level can be characterized as having small sets of language-independent conceptual elements ... and conceptually primitive relationships ... out of which all expressible concepts can be constructed.“ [Bra85], p. 204f
- ***Linguistic level:*** “Finally, going one step higher, we might consider networks whose primitive elements are language-specific. ... This linguistic level represents perhaps the most radical view of semantic nets, in that the >>primitives<< are language-dependent, and are expected to change in meaning as the network grows. Links in linguistic level networks stand for arbitrary relationships that exist in the world being represented.“ [Bra85], p. 205

1. Semantical networks are sets of nodes connected by links.

The first two levels, the implementational and logical level, can roughly be identified with what Newell called the symbol level. They deal with the question how to represent knowledge as a computer program or with a logic-oriented knowledge representation formalism. The other two levels correspond to Newell`s knowledge level. They describe knowledge by pointing out semantical or conceptual relationships of the world or natural languages.

In addition, Brachman identifies a ***missing*** level, which he calls the epistemological level.

- ***Epistemological level:*** "The formal structure of conceptual units and their interrelationships as *conceptual units* (independent of any knowledge expressed therein) forms what could called an *epistemology*. ... The epistemological level of semantic network permits the formal definition of knowledge-*structuring* primitives, rather than particular knowledge primitives." [Bra85] p. 205f. Brachman locates this level between the conceptual and logical level.

The epistemological level fills or narrows the gap between the implementational/logical and the conceptual/linguistic level by defining knowledge and knowledge-structuring primitives. This level is related to the conceptual level because it defines the primitives which can be used at the conceptual level to express knowledge. This level is related to the logical level, because it formally defines these primitives.

A comparable level can be introduced to overcome the ***dualism of symbol level and knowledge level***. This level is to be used to define the epistemologically adequate modelling primitives for knowledge specification at the knowledge level, as proposed by the KADS-I project (cf. [WSB92]), for example. These primitives are to become a declarative semantics to overcome the shortcomings of informal descriptions, like ambiguity, and the lack of techniques for checks for correctness and completeness. Languages which are related to this level are therefore to have the main features:

- In [BrW89], [WSB92] or [Cla92a] there are several language primitives defined which can be used to express different kinds of knowledge and their relationships. The languages must contain these primitives in order to allow the formulation of the model of expertise at the appropriate level.
- The languages must have a formal, i.e. declarative, semantics to allow an unequivocal and well-defined formulation of the model of expertise.
- The languages must have an operational semantics to allow the evaluation of the model of expertise by prototyping.

Languages which fill the gap between the symbol level and the knowledge level must be related to the symbol level by constituting a specific and executable knowledge representation formalism. These languages are related to the knowledge level, because they provide the modelling primitives which are identified as necessary means for describing expertise at this level. They have to overcome the weakness of originally proposed knowledge representation formalisms. Examples for such languages which fulfil the given criteria more or less are: FORKADS (cf. [Wet92]), KARL (cf. [AFS93]), K_{BS}SF (cf. [JoS92]), $(ML)^2$ (cf. [HaB92]), Model-K (cf. [KaV92]), MoMo ([WVL92]), OMOS (cf. [Lin92a]), and QIL ([ARS92]).[1] Every one of these languages defines primitives which allow the explicit representation of

1. Cf.[FeS92], [FeH93], or chapter 6 of the thesis for a comparison of these languages.

what is realized to be relevant for a knowledge level description of a system like the used domain knowledge, the single inference steps and their interaction, and a kind of control knowledge which determines the ordering of the inferences. Additionally, some of these languages define a formal and / or operational semantics for these descriptions. And most of these languages abstract from implementational details which are concerned with an efficient realization at the symbol level. Apart from their common motivations, three ***different*** classes of languages can be identified:

- Languages which can be used to ***operationalize*** models of expertise, like MODEL-K, MoMo, and OMOS. These languages add the advantages of prototyping (i.e., the evaluation by a running program) to the modelling process. A language like MODEL-K includes the modelling primitives of KADS. The knowledge engineer can use frames, Lisp, or Prolog to implement a semantics for them.
- Languages which can be used to ***formalize*** models of expertise like $K_{BS}SF$, $(ML)^2$, and QIL. These languages allow a precise formulation of a model of expertise and enable checks for consistency, redundancy, and correctness by verification techniques. These languages define a formal semantics (mostly a model-theoretical semantics) for the modelling primitives, and the knowledge engineer can declaratively describe the knowledge without implementing it. This means that he is not forced to implement a specific computer agent which computes a solution but he describes a general agent who solves the problem. Because of the definite semantics of these languages this is done in a unique and precise manner.
- Languages which attempt ***both***, like FORKADS[1] and KARL. These languages try to integrate the advantages of formal and operational languages. The main features of KARL[2] (cf. [FAL91], [AFS93]) are:

  • KARL provides modelling primitives which allow the explicit representation of different kinds of knowledge. For this purpose the model of expertise as defined in the KADS project (cf. [WSB92]) were used as a framework and extended by ***hierarchical refinement.*** Complex specifications, acquired for real-life systems, tend to be complicated and difficult to survey. Formal specifications are hard to understand. Therefore, ***graphical representations*** with defined semantics are included in KARL. These modelling primitives ***abstract*** from all details which are concerned with the implementation of the modelled expertise (cf. section 5.3.2).

  • By having a model theoretic semantics, KARL allows a ***formal specification*** of the model of expertise, which helps to avoid ambiguities and permits to reason about the modelled expertise in a formal manner. It provides first-order logic (syntactically extended by object-orientation, attribute inheritance, etc.) which permits modelling the domain layer and the inference layer. For the task layer a restricted variant of dynamic logic (cf. [Koz90]) is given.

---

1. FORKADS does not have a declarative semantics.

2. KARL is designed as an ***intermediate*** representation, which narrow the cognitive distance between the conceptual description of a system and its implementation, and KARL is designed as the technical core of a ***mediating*** representation, which facilitates the communication between domain expert and knowledge engineer (cf. [FBA93]). The use of KARL as part of a mediating representation is discussed in [NeM93] which define a Hypermodel which is embedded in a hypermedia environment and which defines a semiformal knowledge representation for the early knowledge acquisition phase.

• This formal specification can be executed to evaluate the specified expertise by ***explorative*** and ***experimental prototyping*** (cf. [Flo84]). There is no need to program the single inference actions, i.e. to add “symbol level”-specific control knowledge. The declarative descriptions of the elementary inference actions which describes the “what” of the elementary inference actions is used for their evaluation. The realization of the elementary inference actions by efficient algorithms and data structures is related to a later stage in the development process of a knowledge-based system, i.e. to design and implementation.

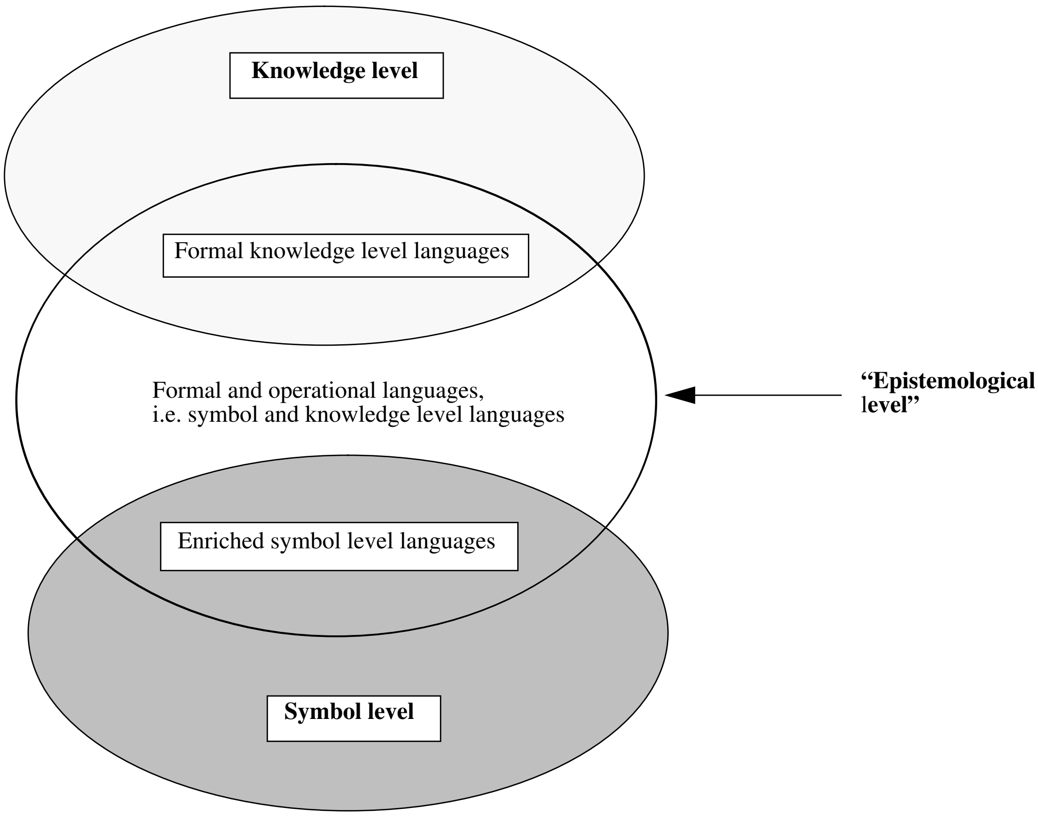


Figure 3. The three levels and the trinity of the third level

These three different classes of languages define different ways of uniting the symbol level and the knowledge level. Languages of the first class can be viewed as an epistemological improvement of the symbol level. They enrich implementation formalisms like Lisp or PROLOG by the modelling primitives of the model of expertise according to the KADS-I project but always require the implementation of a computer program to define their semantics. The other extreme are formal specifications languages like $(ML)^2$. They allow to define a definite semantics for these modelling primitives without specifying a specific computational agent as problem solver. By applying modifications of first-order logic[1], they provide even more expressive power than Turing-completeness to specify a model of expertise at the knowledge level. An intermediate language like KARL has the two main features:

1. $(ML)^2$ uses first-order logic, meta-logic, and quantified dynamic logic.

- It restricts the expressive power to Turing-completeness, which allows the execution of a model of expertise and therefore prototyping.
- It allows the declarative description of a model of expertise, i.e. the description of a general agent which solves the problem instead of requiring the implementation of a computer program, i.e. of a specific agent.

These three classes of languages are rather complementary than contradictory. In the first step a language like $(ML)^2$ can be used to formalize a model of expertise. For the translation of an informally specified expertise into a formal representation it is helpful to have as much expressive power as possible. A language like KARL can be used in a second step to add the flavour of executability to the specified expertise. The price for this is the restriction of the expressive power and richness of modelling constructs of the language. A language like MODEL-K can help to add the aspect of efficiency to the specified expertise by implementing the inference actions. This would help to support the change over to design and implementation without destroying the conceptual structure of the modelled expertise.

1.1.4 The Negation of the Negation

The question may arise if the distinction of symbol level and knowledge level has led anywhere - after all we have come back to discussing knowledge at the symbol level again. But what seems to be an outright contradiction, can as a matter of fact be seen as the way how science works. ***Two sides, i.e. the symbol level and the knowledge level, are divided to allow a better insight in both, which, in turn, allows a reunion of both at a higher and more appropriate level.*** It was Hegel who distinguished two different kinds of negation as a common principle for several development processes.

> "All that is necessary to achieve scientific progress ... is the recognition of the logical principle that the negative is just as much positive, or that what is self-contradictionary doesnot resolve itself into a nullity, into abstract nothingness, but essentially only into the negation of its *paricular* content, in other words, that such a negation is not all and every negation but the negation of a specific subject matter which resolves itself, and consequently is a specific negation, and therefore the result essentially contains that from which it results; which strictly speaking is a tautology, for otherwise it would be an immediancy, not a result. Because the result, the negation, is a *specific* negation it has a *content*. It is a fresh Notion but higher and richer than its predecessor; for it is richer by the negation or opposite of the latter, therefore contains it, but also something more, and is the unity of itself and its opposite. It is in this way that the system of Notions as such has to be formed - ..." [Heg1812]

In a first step, a progress is made by negating a current situation. During this phase, the new entity is mainly discussed under the point of view what it is ***not***. The knowledge level as introduced by Newell was characterized only very weakly, and confusion arose about what the knowledge level is not. Among other things, it was discussed (see the discussion in the next subsections)

- whether knowledge-level descriptions should ***not*** specify how a solution of a problem is reached, and

- whether knowledge-level descriptions should ***not*** use any symbols or logic.

In the course of the further discussions and research, the knowledge level got its own syntactical and semantical meaning by the discovery of generic problem-solving methods and different types of knowledge and primitives to express them. These positive characterizations (i.e., the negation of the negation) have helped to overcome some artificial contradictions of symbol level and knowledge level which were necessary to fix the knowledge level as a new entity, but are no longer necessary after a positive characterization of it has been reached.

1.1.5 "How" and "What" and the Knowledge Level

In [RaP92] it is argued that a knowledge level description should "not be used as a functional specification ... knowledge level descriptions are decoupled from functional specifications." [RaP92], p. 601f. It should only be used to describe the behavior and interaction of the expert during problem-solving and not ***how*** the knowledge-based system would work to solve the problems. This position can also be found in the literature of software engineering. The separation of specification and design is discussed as a separation of "what" and "how" (cf. [Pre87], [Som89], [ThD90], [YoC79], [You89]).

> "The generation of system-level requirements is, to the extent possible, a pure >>what<<, addressing the desired characteristics of the complete system. The next steps, determining the next level of the hierarchy and allocating system requirements to the elements, are in fact a >>how<<." [Dor90], p. 12

During the specification phase it is established in interaction with the users what the system should do. During design and implementation it is realized how the system is to do its tasks, i.e. which algorithmic solution can be applied. This separation[1] does ***not*** work in the same way with the domain of knowledge-based systems (therefore we disagree with the position of [RaP92]) because a high amount of the problem-solving knowledge, i.e. "how" to meet the requirements, is not a question of efficient algorithms and data structures but exists as domain-specific and task-specific heuristics as a result of the experience of an expert. [Par86] uses the concept of "***incompletely specified functions"*** to describe the features of problems only solvable by using knowledge-based systems. For many problems which are completely specifiable (like chess, for example) it is not possible to find an algorithmic solution. The task is easy to specify (e.g., choose the best move) but it is not possible to derive an efficient algorithm from the specification. For the design and implementation of a solution, domain-specific heuristics or domain-specific inference knowledge is needed. "In simple terms this means analysis is not simply interested in ***what*** happens, as in conventional systems, but with ***how*** and ***why"*** [Bro86], p. 323

The contradiction can be solved by distinguishing two different kinds of "how" to solve the problem. "There is a difference between what we would call respectively ***knowledge-level control*** and ***symbol-level control***." [Sch92a], p. 12

1. Even in the domain of software engineering or information system development this separation is often not practicable.

- At the knowledge level there is a description of the domain knowledge and the used problem-solving method which is required by an agent to solve the problem effectively and efficiently.
- At the symbol level there is a description of efficient algorithmic solutions and data structures for implementing an efficient computer program, i.e. a very specific agent.

To give an example: At the knowledge level a single inference step is described declaratively, whereas at the symbol level an efficient algorithmic solution is implemented for the single step. The single steps and their interactions - which also refer to the "how" - are already modelled at the knowledge level.

1.1.6 The Wrong Contradiction of Symbol Level and Knowledge Level

> "Positing the knowledge level as a separate level, the knowledge level has to be different from the symbol level and thus one cannot allow any symbol level constructs at the knowledge level. ... The conclusion then is clear: ... knowledge level models do not exist." [Vin92], p. 621f

It is the merit of [Vin92] to point out the general weakness of the knowledge level as it was introduced by [New82]. Apart from that, the difference of knowledge level and symbol level is not due to the fact that the knowledge level does not use any symbols. The difference is the fact that at the knowledge level a model of an agent is defined which is able to solve the given problem effectively and efficiently, whereas at the symbol level an efficient implementation by an operational computer program is defined. Therefore the question whether logic belongs to the symbol level or knowledge level remains without a definite answer. On the one hand, logic can be used as an implementation language, i.e. Prolog can be used to implement an efficient computer program. On the other hand, logic can be used as a specification language at the knowledge level, especially if its epistemological adequacy is improved by suitable modelling primitives.[1]

1.1.7 Conclusion

It has been the object to complete a discussion which started a decade ago. The separation of symbol level and knowledge level turned out to be quite fruitful because it allowed to describe knowledge independently of its implementation in a specific knowledge representation and inference system. In other respects, however, a lot of confusion arose with this distinction.

It was argued here that this distinction is not a question whether symbols are allowed at the knowledge level or not. With the development of operational and formal knowledge specification languages the gap between both levels was filled. These languages are operational and formal to allow an operational and formal description of knowledge. The languages give epistemologically adequate primitives which allow the expression of a model

1. For the discussion of the two kinds of control knowledge cf. [Sch92a].

of expertise at a high level of abstraction.

It was further maintained here that this distinction is not a distinction in the sense of “what” a system should do at the knowledge level versus “how” a system does this at the symbol level. With every level, a specific part of the “how” is related. At the knowledge level the knowledge is modelled as required by an agent in order to solve a problem effectively and efficiently. The symbol level additionally deals with implementational details necessary for a specific implementation.

1.2 The Different Subtasks of Knowledge Acquisition

The following section will be limited to the description of one step out of the complete knowledge engineering process, which consists of several phases like knowledge acquisition, design, and implementation as proposed by MIKE (cf. [AFL93]). One subtask of the ***knowledge acquisition*** phase, the model construction, will be discussed in more detail. The main task of the knowledge acquisition phase is to build a formal model of the expertise. This model serves as a means for further knowledge acquisitions, thus this phase will be termed model-based knowledge acquisition. Further, as the authors recommend an incremental process model for knowledge acquisition (cf. [AFS90]), the first phase of knowledge engineering will be referred to as model-based and incremental knowledge acquisition. Model-based and incremental knowledge acquisition consists of three main tasks:

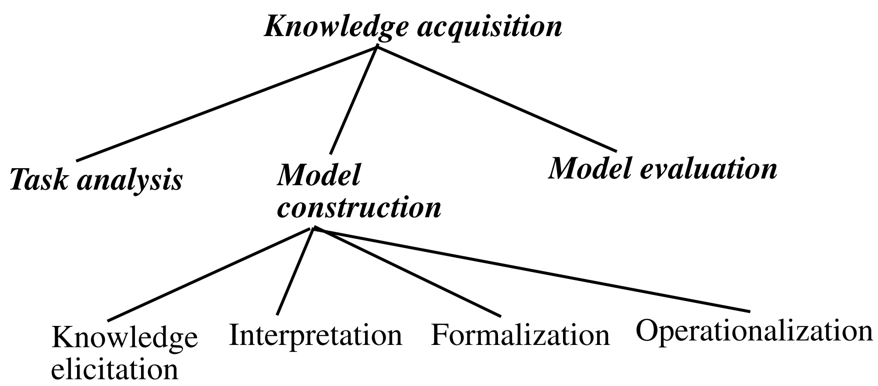


Figure 4. Subtasks of knowledge acquisition

- ***Task analysis***: Within the task analysis the overall functionality of the system to be built as well as that of its environment has to be identified. This means that, apart from the system itself, the environment in which this system is to work, needs to be considered as well.
- ***Model construction***: This subtask will be discussed in the remaining part of the section.
- ***Model evaluation***: This task includes the two subtasks of ***conservative revision of the model*** for deleting semantical and syntactical errors[1] and ***assessment of the model*** for proving its ability to solve the given tasks. Both subtasks are supported by the formal and

operational knowledge specification language KARL, which allows the use of evolutionary prototyping (cf. [Flo84]) during these activities.

The main activity of knowledge acquisition is the construction of a model of the expertise (cf. [WSB92]). This model is an artificial construct, i.e. it does not already exist in the expert´s ability. It is a theory for constructing an executable computer program which will be able to support problem-solving in a specific domain and for specific tasks.[1] The expert´s ability is one of the empirical sources which can be used for its creation. It is not intended to rebuild a cognitively adequate model of the expertise of the expert but rather to describe the expertise of the expert, which is based on implicit knowledge and skill, in an explicit and formal manner (cf. [Ber87]). On the contrary, the modelling process will influence and change the cognitive model of the expert by making him aware of some foundations of his own expertise. The model construction consists of four subtasks (see figure 5).

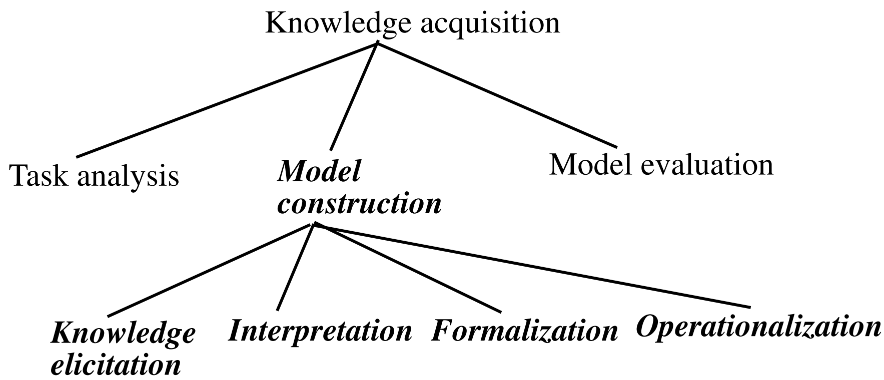


Figure 5. Subtasks of model construction

- ***Knowledge elicitation***: The expert must be interviewed and observed in order to get an empirical foundation for the model of expertise. The data or knowledge are described at the ***linguistic level*** in a natural language. It mainly consists of interview protocols, protocols of verbal reports, etc.
- ***Interpretation:*** These knowledge protocols must be interpreted during the ***interpretation step*** to obtain the represented meanings. The goal is to represent the knowledge at the ***conceptual level***. There is no direct mapping from the linguistic to the conceptual level because different words can stand for the same concept, and the same word can stand for different concepts. During the interpretation step the knowledge engineer has to learn to understand a new language (cf. [Fen92]). The result of this step is a ***model*** which describes the main steps and their interactions which constitute the problem-solving process. In MIKE (cf. [AFL93]) this model is represented in a hypermedia environment (cf. [HoN92]) and used as a ***mediating representation*** which facilities communication between expert and knowledge engineer (cf. [FBA93]).

---

1. MIKE distinguishes between conservative and revolutionary revision of models (cf. [Emd91]). Whereas revolutionary revisions are done by restarting a new cycle of all subtasks of knowledge acquisition, i.e. starting with a new task analysis or knowledge elicitation step, detection and deletion of local errors are done during the model evaluation step.

1. Cf. [Nau85] which takes the view on programming as a theory-building activity.

- ***Formalization:*** This model, which is already described in a restricted and structured natural language, for instance as a hypertext document with predefined kinds of nodes and links, must be formalized during the ***knowledge formalization substep***. A formal specification language which can be used to describe the knowledge at the logical level is KARL. It provides a logic-based language for the description of the domain-specific and the generic parts of an expertise. During formalization, two main results are reached:

  • The formalization defines a technical core of the model which is used as a mediating representation.

  • The formal model defines an ***intermediating representation*** (cf. [FBA93]) between the conceptual model of the knowledge engineer and the final implementation of the system and narrow the cognitive gap between both.

- ***Formalization at the knowledge level:*** In addition, KARL provides predefined modelling constructs which allow a knowledge specification closely related to the conceptual level, but which have the same clear semantics as the logical level. This intermediate level between the logical and the conceptual level is related to what [Bra85] calls the ***epistemological level***. It allows the epistemologically adequate formalized modelling of knowledge, or a formal description of the expertise at the knowledge level.

- ***Operationalization:*** The model should be operationalized in order to allow knowledge evaluation by prototyping. If an operational formal knowledge specification language like KARL has been chosen, the formal specification must be refined to fulfil the stronger restrictions of the operationalization of KARL.[1]

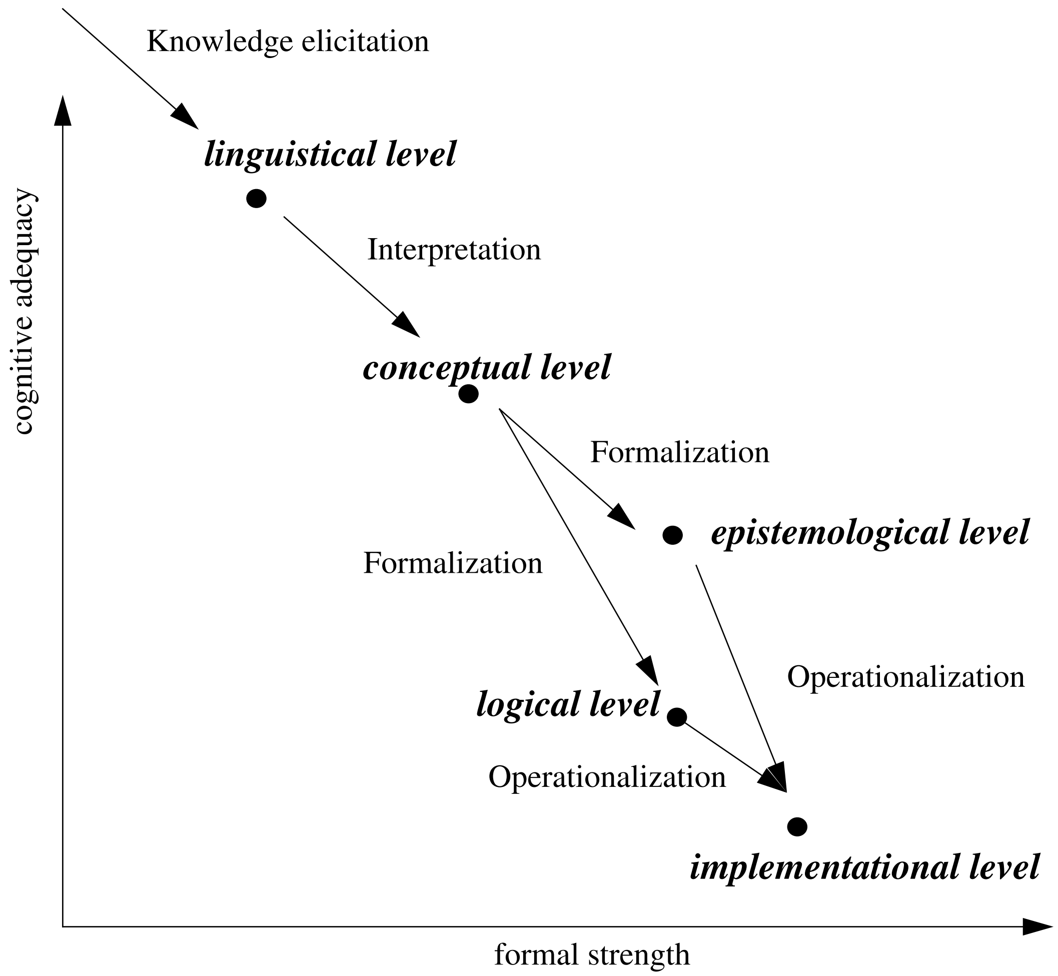


Figure 6. The five levels of knowledge representation (cf.[Bra85])

1. Rules must be safe to prevent infinite models, for example, and the syntax requirements of the operational KARL are more restrictive (cf. [Ang93]).

Chapter 2

KARL at the Formal Level

KARL is not a language but an union of two languages. ***Logical-KARL (L-KARL)*** is used to describe static knowledge. ***Procedural-KARL (P-KARL)*** is used to describe procedural knowledge.

- ***L-KARL***: Because first-order logic is not well suited as a knowledge representation formalism (cf. [Bra87]) but as a foundation for a well-defined semantics of a knowledge representation formalism L-KARL extends first-order logic by the use of modelling primitives which should improve the epistemological adequacy of the language. The logical language L-KARL is a customization of O-logic (cf. [Mai86]) with its significant modifications (cf. [KiW89], [KiW93]) and F-logic (cf. [KiL89], [KLW90]). O-logic and F-logic extend normal first-order logic by using object-orientation and set-valued attributes but preserve first-order semantics, i.e. they do not extend its expressive power.
- ***P-KARL***: Experiences with XCON (cf. [SBJ87], [BaS89]) show that great problems arise if the control flow is only specified implicitly, e.g. in production rules. In KADS, knowledge about the control flow is clearly separated from other kinds of knowledge by describing it on a separate layer of the model of expertise (cf. [WSB92]). So P-KARL is used to explicitly describe knowledge about the control flow of a problem-solving method.

Both languages have a formal, i.e. model-theoretical, semantics to fulfil one claim of KARL. KARL is to be a ***formal*** knowledge specification language which allows a unique, correct, and complete specification of a model of expertise. Both sublanguages and their interactions are modifications of existing logics to fulfil a second claim of KARL, i.e. the specification of knowledge at the ***knowledge level***. Therefore, first-order logic is extended it by the use of object-orientation, resulting in L-KARL, and combined it with a variant of dynamic logic (cf. [Har79], [Har84], [Koz90]), resulting in P-KARL. Both sublanguages and their interaction are discussed in the following. In chapter 3 it is shown how these sublanguages and their interaction are used to model the domain layer, inference layer, task layer, and their connections.

2.1 L-KARL: The Logical Language of KARL

The logical language L-KARL is used to declaratively describe knowledge at the domain layer and the inference layer as well as for the mapping of both layers.

The application of logic as a programming language was first discussed in [Kow74], [EmK76] and has the following main advantages:[1]

- First-order predicate logic[2] has a well-defined model-theoretical semantics. Its application therefore provides an exact and well-defined meaning of a specification.
- The procedural interpretation of Horn logic by the use of minimal model semantics allows to simulate the computation of results similar to other executable programming languages. This allows the use of L-KARL for operationalization and prototyping (cf. [Ang93]). The use of logic as a programming language by applying a minimal model semantics is discussed in [Apt90], [Prz88], [Llo87], [Ull88]. Especially [Llo87] has become standard literature in regard to logic programming.

O-logic and F-logic extend normal first-order logic through the use of object-orientation and set-valued attributes but preserve first-order semantics. The logical language L-KARL is a customization of ***O-logic*** (cf. [Mai86]) with its significant modifications (cf. [KiW89], [KiW93]) and of ***Frame logic*** (F-logic, cf. [KiL89], [KLW90]). The reasons for choosing these kinds of formalisms are:

- It has a formal semantics, i.e. a model-theoretical one. A specification using the logical language L-KARL therefore has a well-defined semantics. F-logic is a language where concepts like attribute inheritance or well typing are provided with a formal semantics.
- The introduction of objects[3] improves the epistemological adequacy of the language and supports object-oriented or frame-based design and implementation of the final expert system. A paradigm shift is prevented by passing from the analysis phase to the design phase.
- O-logic and F-logic introduce the explicit representation of sets without destroying the first-order semantics of the language.
- F-logic allows formal reasoning about classes. Therefore, not only sets of individuals but also sets of classes become possible answers of a query.

---

1. In addition, first-order logic has been broadly applied in the domain of deductive data bases (cf. [CGT89], [Ull88], [Ull89]).
2. For a general introduction to first-order predicate logic see [End72], [HoK91], [NiM90], [SpA91].
3. Cf. [Kim90] for an introduction to object-orientation.

The contents of this chapter are organized as follows. The first section discusses the main ideas of other approaches which are used to define the model-theoretical semantics of L-KARL. Then, syntax, informal semantics, general model-theoretical semantics, and Herbrand semantics of L-KARL are discussed in great detail. Even L-KARL is “only” a customization of F-logic, its semantics must be defined according to the restrictions and modifications of F-logic which are defined by L-KARL.

Minimal Herbrand model semantics and perfect Herbrand model semantics, as developed in the domain of logic programming (cf. [Apt90], [Llo87]) and deductive databases (cf. [Prz88], [Ull88]), can be applied in a straightforward manner to L-KARL. Therefore, these types of semantics are discussed in less detail and it is assumed that the reader is familiar with the idea of minimal models.

Finally, the use of constraints and built-in predicates in L-KARL are described and a brief comparison of the language primitives of L-KARL with O-logic and F-logic is added. It is shown that L-KARL defines an intermediate point between O-logic on the one hand and F-logic on the other hand.

2.1.1 Significant Ideas of Other Approaches Used for L-KARL

L-KARL applies several ideas from the domain of logic programming (cf. [Apt90], [Llo87]) and deductive and object-oriented databases (cf. [Prz88], [Ull88], [Bee90], [KLW90], [KiW93]) for its model-theoretical semantics. In the following, the work which was used to define a declarative semantics for L-KARL is sketched.

Object-orientation and equality

One significant feature of L-KARL is the integration of object-orientation into a declarative logic-based framework. The key-point is the question as to how one can express the derivation of “new” objects or object denotations. The early version of O-logic (cf. [Mai86]) failed to define a declarative semantics for this concept. O-logic according to [KiW89] or [KiW93] and F-logic (cf. [KiL89], [KLW90]) express this applying function symbols. If an object “o” is given and a new object should be derived, this successor can be expressed by “new(o)”, i.e. by applying a function to the constant which denotes the object “o”. As a consequence, it is often necessary to express the fact that different object denotations denote the same object, i.e. syntactically different object denotations are semantically equivalent. This requires a modification of the standard definition of a Herbrand interpretation according to Herbrand semantics of logic programs with equality (cf. [SpA91], [KLW90]).

Reasoning about classes

The first version of KARL as published in [FAL91] could not formally reason about classes, i.e. predicates. In the case of a classification problem the final classes had to be modelled as constants to allow them to be attached to variables as a result of a query. But modelling individuals and terminal classes using constants and superclasses using predicates is very artificial. It does not reflect a modelling decision but is an artefact of the modelling language. The same problem would arise in order-sorted logical languages. The sorts define hierarchical domain restriction for the variables but it is not possible to instantiate a variable directly by a type, i.e. the sort hierarchy cannot by used to express a domain hierarchy which should be browsed. In $(ML)^2$ (cf. [HaB92]) this problem is solved by applying meta-logic. Arbitrary terms of an object-logic can be used as constants by the meta-logic. In F-logic, and in the same manner in L-KARL, this is reached by semantically handling (i.e., interpreting) classes as individuals. A class is interpreted by an object in the same way in which an element of a class is interpreted by an object. The following clause should illustrate this. The predicate "classes_of(element : _, class : _)" collects all classes of which an object is an element.

$\forall x$ (classes_of(element : o, class : x) $\leftarrow$ o ε x).

Integration of well-typing into a model-theoretical semantics

Well-typing defines domain and range-restrictions for attributes. An interpretation is well-typed if it does not contain the application of an attribute for an object for which the attribute is not defined and if it does not contain a value of an attribute which is not an element of the classes which define the range of the attribute. Instead of external type-checking with a procedural semantics, F-logic integrates these well-typing conditions into the model-theoretical semantics of a set of clauses. The following given interpretation function $I_{\Rightarrow}$ is used to define a model-theoretical semantics for domain restrictions and range restrictions (cf. [KLW90], p. 14):

$I_{\Rightarrow}$ interprets each attribute "a" as a ***partial anti-monotonic*** function

$I_{\Rightarrow}(a): U \rightarrow \wp\uparrow(U)$

where $\wp\uparrow(U)$ denotes all ***upwardly-closed subsets*** of U.

$I_{\Rightarrow}(a)(o) = \{c_1,..., c_n\}$ is the set of classes which define the range restriction for the attribute "a" applied to the object "o". The reasons why these functions have to be partial anti-monotonic and why the ranges of the functions have to be upwardly-closed subsets will be given in section 2.1.4.

$I_{\rightarrow}$ interprets each attribute "a" as a ***partial*** function

$I_{\rightarrow}(a): U \rightarrow U$.

$I_{\rightarrow}(a)(o)$ is the attribute value of "a" applied to the object "o". An interpretation is well-typed if it fulfils the following two requirements (cf. [KLW90], p. 15):

- Domain restriction: if $I_{\rightarrow}(a)(o)$ is defined then so is $I_{\Rightarrow}(a)(o)$;
- range restriction: if $I_{\rightarrow}(a)(o)$ is defined then $I_{\rightarrow}(a)(o) \leq c$, for all $c \in I_{\Rightarrow}(a)(o)$.

Set-valued attributes

O-logic and F-logic allow sets as values for an attribute. This feature are integrated into L-KARL because it was a very natural way to model domain features and results of a problem-solving process for several applications.

Minimal and Perfect Models as Semantics

Minimal models as a semantics for logical programs was introduced by [Kow74] and [EmK76]. Because F-logic and L-KARL are just syntactical modification of predicate logic, these kinds of semantics can easily be applied to them. The perfect model semantics is a generalization of the minimal model semantics. Whereas a unique minimal models only exists for sets of Horn clauses, perfect models exist for sets of stratified clauses. Sections 2.1.6 and 2.1.7 sketch these two semantics and their application. The reader is further referred to [Llo87], [Ull88], and [Apt90].

L-KARL and F-logic

L-KARL is a customization of F-logic. F-logic is a powerful formal language for investigating the integration of object-orientation, functional and set-valued attributes, well-typing, and attribute inheritance into a well-defined semantical framework. Therefore, the semantics of F-logic was used for defining a semantics for L-KARL. On the other hand, F-logic is more a framework for declarative and object-oriented languages than a modelling language by itself.

- F-logic makes less epistemological distinctions. It does not distinguish between elements and classes and, e.g., therefore does not distinguish between is-a and is-element-of relationships.
- F-logic allows a dynamic change of is-a hierarchies, inheritance of attributes, and domain restrictions and range restrictions for attributes. Compared with order-sorted logic this would correspond to modifications of a sort hierarchy as a result of the truth of a fact. L-KARL excludes this property of F-logic to prevent modifications of terminological knowledge as a result of case-specific input data. The terminology at the domain layer and inference layer is considered as fixed, i.e. it should only be changed as a result of an explicit knowledge acquisition steps.[1]

The reader is also referred to section 2.1.10 which compares L-KARL with O- and F-logic.

2.1.2 Syntax of L-KARL

In the next definition, an alphabet of an L-KARL language is introduced. It contains all elements which are used to build expressions of an L-KARL language. It consists of element, class, and value denotations (i.e. numbers and strings), attribute and argument names, predicate symbols, variables, logical connectives and quantifiers, and some additional letters which will be discussed further on.

Def. 1 Alphabet of an L-KARL language

An ***alphabet*** (E, Σ, Boolean, Integer, Real, String, $A_{E\rightarrow}$, $A_{\Sigma\rightarrow}$, $A_{E\rightarrow\rightarrow}$, $A_{\Sigma\rightarrow\rightarrow}$, Π, A_{Π}, X_E, X_{Σ}, X_V) of an L-KARL language consists of:

- A finite set E of ***element denotations***. The elements of E are function symbols with a fixed arity k, $0 \leq k$. 0-ary symbols in E are called constants.
- A finite set Σ of ***class names***.

1. Future versions of KARL will probably use this feature of F-logic to enhance the mapping of the domain layer on the inference layer.

- The set Boolean = {False, True}, an infinite set Integer of all ***integers***, an infinite set Real of all ***real numbers,*** and an infinite set String of all ***strings with finite length***. The union of these sets defines all ***values*** and is denoted by ***V***. Elements of String are enclosed by quotes. These sets are called ***value types***.
- Two finite sets $A_{E\rightarrow}$ and $A_{\Sigma\rightarrow}$ of ***single-valued attributes*** and two finite sets $A_{E\rightarrow\rightarrow}$ and $A_{\Sigma\rightarrow\rightarrow}$ of ***set-valued attributes***.
- A finite set $\prod$ of ***predicate symbols***.
- A finite set $A_{\prod}$ of ***argument names***.
- Three finite sets X_E of ***element variables***, X_Σ of ***class variables***, and X_V of ***value variables***.
- ***Logical connectives*** "$\wedge$", "$\vee$", "$\neg$", "$\leftarrow$” and ***quantifiers*** "$\exists$", "$\forall$“.
- Additional alphabet symbols are: “CLASS”, “CLASS_ATT”, “ELEMENT_ATT”, “END”, “ISA”, “PREDICATE”, “≤”, “ε”, “≅”, “(“, “)”, “[“, “]”, “:”, “::”, “{“, “}”, “ “, “,” “;”, and “.”.

The sets E, Σ, V, $A_{E\rightarrow}$, $A_{\Sigma\rightarrow}$, $A_{E\rightarrow\rightarrow}$, $A_{\Sigma\rightarrow\rightarrow}$, $\prod$, $A_{\prod}$, X_E, X_Σ, and X_V are disjoint.

In the next step, ***ID terms*** are defined which can be used to denote objects (E-ID terms), classes (C-ID terms), and values (V-ID terms). These ID terms correspond to terms in predicate logic.

Def. 2 ID terms

An ID term is either an E-ID term, a C-ID term, or a V-ID term.

- An ***E-ID term*** is a term composed out of element denotations (i.e. function symbols and constants) which are element of E and element variables which are elements of X_E. These terms are constructed in the way as it is known from predicate logic.
- E* denotes the set of all ground[1] E-ID terms.
- A ***C-ID term*** is either an element of Σ or X_Σ.
- A ***V-ID term*** is either an element of V or X_V.

Examples for E-ID terms are f(g(a,x)) or h(h(h(a,x),y),z).

A ***class term*** can be used to define attributes applicable to the class and its elements. A class attribute, i.e. an element of $A_{\Sigma\rightarrow}$ or $A_{\Sigma\rightarrow\rightarrow}$, is applicable to the class and an element attribute, i.e. an element of $A_{E\rightarrow}$ or $A_{E\rightarrow\rightarrow}$, is applicable to the elements of the class. In the next section an example is given which should clarify this topic.

1. See Def. 10 “Ground term and ground formulae” on page 52

Def. 3 Class term[1]

A ***class term*** is a term

c[[

CLASS_ATT

$ca_1 : \{ca_{11},..., ca_{1k_1}\};$

$...;$

$ca_r : \{ca_{r1},..., ca_{rk_r}\};$

$cs_1 :: \{cs_{11},..., cs_{1l_1}\};$

$...;$

$cs_t :: \{cs_{t1},..., cs_{tl_t}\};$

ELEMENT_ATT

$ea_1 : \{ea_{11},..., ea_{1m_1}\};$

$...;$

$ea_u : \{ea_{u1},..., ea_{rm_u}\};$

$es_1 :: \{es_{11},..., es_{1n_1}\};$

$...;$

$es_v :: \{es_{v1},..., es_{vn_v}\};$

]]

The following conditions must hold true:

- The class name "c" is an element of Σ.
- The ca_{gh}, cs_{gh}, ea_{gh}, and es_{gh} are elements of Σ or value types.
- Further holds: $ca_i \in A_{\Sigma\to}$, $ea_i \in A_{E\to}$, $cs_j \in A_{\Sigma\to\to}$, and $es_j \in A_{E\to\to}$.

An ***is-a term*** defines that a class is a subclass of another, i.e. every element of the first class is an element of the second class and every attribute which is defined for the second class is inherited by the first class.

Def. 4 Is-a term

An ***is-a term*** is a term

$c \leq c'$

where "c" and "c´" are C-ID terms.

A ***class definition*** is a conjunction of a class term and corresponding is-a terms. It defines attributes which are applicable to a class, its subclasses, and their elements.

1. In order to distinguish semantically between class denotations for which a class term exist and class denotations for which a class term does not exist an attribute definition "exists : {}" is associated with every class term. This ensures that a class term "c[[]]" is not a tautology, cf. Def. 21 "Satisfaction of a class term" on page 71. This technical detail is hidden to the user of L-KARL (cf. [KLW90], p. 52). It is required to define an output of an elementary inference action, cf. Def. 110 "Output of an elementary inference action" on page 160.

Def. 5 Class definition

A ***class definition*** is a term

$c[[...]] \wedge c \leq c_1 \wedge ... \wedge c \leq c_n$

where c[[...]] is a class term and all c_i for i=1,...,n are elements of Σ (i.e., they are ground C-ID terms).

Remark 1 Alternative syntax for class definitions

Instead of

c[[
CLASS_ATT
$ca_1 : \{ca_{11},..., ca_{1k_1}\};$
...;
$ca_r : \{ca_{r1},..., ca_{rk_r}\};$
$cs_1 :: \{cs_{11},..., cs_{1l_1}\};$
...;
$cs_t :: \{cs_{t1},..., cs_{tl_t}\};$
ELEMENT_ATT
$ea_1 : \{ea_{11},..., ea_{1m_1}\};$
...;
$ea_u : \{ea_{u1},..., ea_{rm_u}\};$
$es_1 :: \{es_{11},..., es_{1n_1}\};$
...;
$es_v :: \{es_{v1},..., es_{vn_v}\};$
$]] \wedge c \leq c_1 \wedge ... \wedge c \leq c_n$

an alternative way to write a class definition is:

CLASS c
ISA $c_1,..., c_n;$
CLASS_ATT
$ca_1 : \{ca_{11},..., ca_{1k_1}\};$
...;
$ca_r : \{ca_{r1},..., ca_{rk_r}\};$
$cs_1 :: \{cs_{11},..., cs_{1l_1}\};$
...;
$cs_t :: \{cs_{t1},..., cs_{tl_t}\};$
ELEMENT_ATT
$ea_1 : \{ea_{11},..., ea_{1m_1}\};$
...;
$ea_u : \{ea_{u1},..., ea_{rm_u}\};$
$es_1 :: \{es_{11},..., es_{1n_1}\};$
...;

$es_v :: \{es_{v1},..., es_{vn_v}\};$

END

F-terms, which are defined by the following definition, can be used to define logical relationships between ID terms. An ***is-element-of term*** defines that an element which is denoted by an E-ID term is an element of a class which is denoted by a C-ID term. An ***equality term*** is used to indicate that syntactically different ID terms denote the same object or value, i.e. it indicates their semantical identity. A ***data term*** is used to describe an object, i.e. an element or a class, by attribute values. These attribute values can be single-valued or set-valued.

Def. 6 F-term

An ***F-term*** is either

- an ***is-element-of term***
  $x \ \varepsilon \ c$[1]
  where "x" is an E-ID term and "c" is a C-ID term.
- an ***equality-term***
  $o \cong o'$
  where "o" and "o´" are both ID terms.
- or a ***data term***
  $o[a_1 : T_1,..., a_r : T_r, s_1 :: \{S_{11},..., S_{1m_1}\},..., s_t :: \{S_{t1},..., S_{tm_t}\}]$
  - where "o" is an E-ID term, $a_i \in A_{E\rightarrow}$ for i=1,...,r, $s_j \in A_{E\rightarrow\rightarrow}$ for j=1,...,t,
  - or "o" is a C-ID term, $a_i \in A_{\Sigma\rightarrow}$ for i=1,...,r, $s_j \in A_{\Sigma\rightarrow\rightarrow}$ for j=1,...,t,
  - all T_i and S_{ij} are arbitrary data terms or V-ID terms in both cases.

There are two kinds of data terms. First, data terms can be used to describe elements using the values of functional and set-valued attributes defined for elements. Second, data terms can be used to describe classes using the values of functional and set-valued attributes defined for classes. As mentioned above, values are not described by attributes. Therefore, no data terms exist which describe values, i.e. "o" is an E-ID term or a C-ID term.

A ***predicate definition*** defines names for the arguments of a predicate in order to improve readability of predicate expressions and domains for the arguments, i.e. these predicates are typed. Not every predicate symbol must have a definition.

Def. 7 Predicate definition

A ***predicate definition*** is a term

PREDICATE p

$p_1 : \{c_{11},..., c_{1r_1}\};$

...,

1. We use "ε" instead of "∈" in order to distinguish between the element "ε" of the alphabet of a L-KARL language and its interpretation by the meta language symbol "∈". For similar reason we use "≅" instead of "=" and "≤" instead of "⊆".

$p_k : \{c_{k1},..., c_{kr_k}\};$

END

The following conditions must hold:

- "p" is a k-ary ($0 \leq k$) predicate symbol, i.e. an element of $\prod$.
- all p_i are argument names, i.e. elements of $A_{\prod}$.
- all c_{gh} are element of Σ or value types.

A ***P-term*** corresponds to a predicate as is familiar from predicate logic. The difference is that the terms of a predicate need not be flat, i.e. a term of a predicate can be a data term with attributes, and that the arguments are named.

Def. 8 P-term

A ***predicate term (P-Term)*** is an expression:

$p(p_1:T_1,..., p_k:T_k)$

- where "p" is a k-ary ($0 \leq k$) predicate symbol, i.e. an element of $\prod$,
- p_i is an argument name (i.e., element) of $A_{\prod}$,
- and all T_i are data terms or V-ID terms.

In the next definition ***formulae*** are introduced.

Def. 9 Formulae

An ***elementary formula*** is either an is-a term, an F-term, or a P-term. ***Composed formulae*** are built up from them using logical connectives or quantifiers as it is familiar from first-order logic (cf. [End72], [Llo87]).

Def. 10 Ground term and ground formulae

A ***ground term*** is a variable-free term. A ***ground formula*** is a variable-free formula.

For the sake of comfort the term ***definition*** is defined.

Def. 11 Definition

A ***definition*** is either a class definition or a predicate definition.

The following definition sums up the above and defines a ***L-KARL language***.

Def. 12 L-KARL language

A ***L-KARL language*** given by a L-KARL alphabet is the set of all formulae and definitions which can be constructed from the symbols of the alphabet according to the above definitions.

Remark 2 Short-cuts for data terms

- Instead of
  $o \,\varepsilon\, c \wedge o[a_1 : T_1,..., a_r : T_r, s_1 :: \{S_{11},..., S_{1m_1}\},..., s_t :: \{S_{t1},..., S_{tm_t}\}]$
  one can write
  $o[a_1 : T_1,..., a_r : T_r, s_1 :: \{S_{11},..., S_{1m_1}\},..., s_t :: \{S_{t1},..., S_{tm_t}\}] \,\varepsilon\, c$
- Empty brackets “[]” can be omitted if it does not cause an error.
- A value term “v” is sometimes written as an empty data term v[].

2.1.3 Informal Semantics of L-KARL

L-KARL is a first-order language with some syntactical extensions. These extensions are intended to simplify the modelling process in KARL and reflect the special purpose for which KARL is designed. KARL is designed for a formal modelling process at the knowledge level. This has two implications:

- L-KARL abstracts from all implementational details which are necessary to realize an efficient program at the so-called symbol level. KARL and L-KARL are not designed to tell a computer how to compute a solution efficiently but they are designed to allow the specification of knowledge in a formal, i.e. definite, consistent, and complete manner.
- On the one hand, the formal semantics of L-KARL lacks the elegance of the formal semantics of first-order logic or F-logic and also does not extend their expressive power. On the other hand, L-KARL customizes these languages to close the gap between formal languages and the conceptual level. Objects, values, classes, attributes, and their functional dependencies (e.g., that a person cannot have two ages), attribute inheritance and well-typing can be directly represented in L-KARL. This customization also provides the possibility of representing L-KARL primitives graphically to improve their understandability. Note, that only the core language is discussed and not the language the user is really working with. In section 5 of the thesis this topic is discussed in more detail. The main advantages of L-KARL is that these graphical modelling primitives receive a formal and well-defined semantics.

Therefore KARL and L-KARL are designed to close the gap between the informal or semiformal knowledge representation by natural language on the one hand and computer readable knowledge implementations on the other hand. In the following, an informal semantics of the modelling primitives is provided before the formal semantics is defined in the next sections. An example dealing with the domain of scientists is incrementally developed according to the different modelling primitives of L-KARL.

2.1.3.1 ID term

The basic modelling primitives in L-KARL are ***elements***, ***classes***, and ***values***, i.e. the ***denotations*** of them. For this purpose the sets E, Σ, and V are given. Elements and classes are considered as real-world entities and modelled by objects. These allow description and reasoning about elements and classes. Values are considered as features of real-world objects and are not modelled as objects by themselves. Therefore, elements and classes can be described by attributes. Values can only be used to describe elements or classes. Clearly it depends on the domain-specific and task-specific view of a modeller in respect to what he wants to model as an object or as an ordinary value. "20" can be considered as the value for an "age"-attribute or it can be considered as a natural number which has the property of being even. Then an object corresponding to the number "20" must be defined.

The set of element constructors E contains constants and function symbols. A constant denotes an object in the universe of an interpretation of the logical language. The function symbols can be used to create new object denotations in dependency on given ones. For example, the constants

MKo, JAn, RSt, SNe $\in$ E

can be used to denote persons, i.e. real world elements. To denote a real world object in dependency on others the unary function symbol

supervisor(_) $\in$ E[1]

can be used. supervisor(JAn) and supervisor(supervisor(JAn)) are two possibilities. The above given constants and the two applications of the unary function symbol supervisor(_) are ground E-ID terms. An example of a non-ground E-ID term would be

supervisor(x_E), $x_E \in X_E$.

E = {MKo, JAn, RSt, SNe, supervisor(_)}
X_E = {x_E,...}

The set of class names Σ only contains constants which can be used to denote real world entities which corresponds to classes of elements, the class of post graduate students, for example.

humans, females, post_graduate_students, professors, scientists $\in \Sigma$

can be used to denote these sets. Contrary to F-logic, L-KARL only allows constants and not functions as elements of Σ. Therefore, the derivation of new class denotations cannot be expressed in L-KARL. The set of classes are considered as fixed. They represent terminological knowledge which should be fixed.[2]

Σ = {humans, females, post_graduate_students, professors, scientists}
X_Σ = {x_Σ,...}

The set of values V only contains constants which can be used to describe other real-world

1. Supervisor of a dissertation project.
2. Therefore, non-ground class terms are not allowed.

objects. It consists of four sets:

- The set Boolean = {False, True} which can be used to model boolean attributes,
- the set Integer of all integers,
- the set Real of all real numbers, and
- the set String of all strings with finite length. Elements of String are enclosed by quotes.

V = Boolean ∪ Integer ∪ Real ∪ String
X_V = {x_V,...}

2.1.3.2 Class Definitions

Besides denoting objects which may represent elements or classes it must be possible to describe features of them. For this purpose the four sets $A_{E\rightarrow}$, $A_{\Sigma\rightarrow}$, $A_{E\rightarrow\rightarrow}$, and $A_{\Sigma\rightarrow\rightarrow}$ are given. $A_{E\rightarrow}$ is the set of all attributes which can be applied to elements, i.e. elements of E. These attributes are ***functional***, i.e. the same element cannot have several values for the same attribute. The set $A_{E\rightarrow\rightarrow}$ contains ***set-valued*** attributes which allow the description of an element by means of a set of values. In our example is defined:

$A_{E\rightarrow}$:= {age, boss, name}
$A_{E\rightarrow\rightarrow}$:= {colleagues, female_colleagues, subjects}

The other two sets also contain single-valued and set-valued attributes. But these two sets define attributes which can only be applied to classes and not to elements. Only one set-valued attribute “all_subjects“ for classes is given:

$A_{\Sigma\rightarrow\rightarrow}$:= {all_subjects}

The applicability of class attributes for classes is defined by class and is-a terms, i.e. by class definitions. The part of the attribute definition of a class definition which is labelled by “CLASS_ATT” defines them. In addition, all attributes of superclasses are inherited. The applicability of element attributes for elements is defined by class definitions and is-element-of terms. They are defined by the part of the attribute definition of a class definition which is labelled by “ELEMENT_ATT”. In addition, all attributes of superclasses are inherited. An element must be an element of the class to allow the application of its attributes.

Let the following class definitions be given:

```
CLASS humans
   ELEMENT_ATT
         age : {Integer};
         name : {String};
END

CLASS scientists
   ISA  humans;
   CLASS_ATT
      all_subjects :: {String};
```

ELEMENT_ATT
boss : {**professors**};
female_colleagues :: {**females**, **scientists**};
colleagues :: {**scientists**};
subjects :: {**String**};
END

The class “humans” has no own class attribute, but all its elements can be described by the attributes “age” and “name”. The class “scientists” has the class attribute “all_subjects” and the element attributes “boss”, “colleagues”, “female_colleagues” and “subjects”. Because it is a subclass of humans, i.e. scientists ≤ humans, it also inherits the element attributes “age” and “name”. “age”, “boss”, and “name” are singled-valued attributes, i.e. an object can only have one value for these attributes. “all_subjects”, “colleagues”, “female_colleagues”, and “subjects” are set-valued attributes, i.e. an object can have several different values for them.

These class definitions define signatures for classes and elements. The application of the attribute “age” to an element which is not a member of “humans” would cause a typing error, because the ***domain restriction*** of the attribute would be violated.

In addition, the class definitions define ***range restrictions*** for attributes, i.e. restrictions for the allowed values of an attribute[1]. Three different cases can be distinguished.

- Value types like Integer or String can be used to restrict the domain of the attribute. This implies that the value of the attribute must be an element of the respectively value type.
- A finite set of classes can be used to restrict the domain of the attribute. This implies that the values of the attribute must be elements or subclasses.
- No range restriction is defined, i.e. the empty list of classes “{}” is used as range restriction.

The values of the attribute “subjects” have to be elements of String, for example. For example, the values of the attribute “colleagues” have to be elements or subclasses of the class “scientists” and the values of the attribute “female_colleagues” have to be elements or subclasses of the classes “scientists” ***and*** “females”.

Besides the inheritance of attributes, the is-a terms specify a ***subset relationship*** between the classes, i.e. between their element sets.

c ≤ c´

implies that every element of “c” is also an element of “c´”. Because “scientists” is a subclass of “humans”, every element of “scientists” is also an element of “humans”. The following two class definitions complete our example.

CLASS **post_graduate_students**
ISA **scientists;**
END
CLASS DEFINITION **professors**
ISA **scientists;**
END
CLASS **females**

1. A value of an attribute can be a value, an element, or a class.

ISA **humans;**
END

2.1.3.3 F-Terms

The next step is to connect element denotations and class denotations by ***is-element-of terms***. In an extension of F-logic L-KARL separates element and class denotations to separate the is-element-of terms from the is-a terms. The first ones model ***element*** relationships whereas the second ones model ***subset*** relationships. For our example, the following ground is-element-of terms should hold:

JAn ε **post_graduate_students**
MKo ε **post_graduate_students**
SNe ε **post_graduate_students**
RSt ε **professors**
supervisor(JAn) ε **professors**
...

Additional, the following relationships can be derived logically:

JAn ε **scientists**
SNe ε **scientists**
RSt ε **scientists**
MKo ε **humans**
...

The ***equality terms*** can be used to express the fact that different E-ID terms indicate the same object. For example:

supervisor(JAn) ≅ **RSt**

indicates that the two different object denotations:

supervisor(JAn)
RSt

denote the same object, i.e. element. Therefore, the ***unique-name assumption*** need ***not*** hold for element denotations. For class names and values the unique-name assumption is assumed, i.e.

$o \cong o'$

and $o, o' \in \Sigma$ or $o, o' \in V$ is always false if $o \neq o'$.[1] This does not mean, that equality term with non-ground C-ID terms or V-ID terms are always false. For example,

$\mathbf{x_\Sigma \cong humans}$

is true, if the variable x_Σ is assigned to the class name "humans".

An equality term

$o \cong o'$

is always false, if "o" and "o´" are not both E-ID terms, C-ID terms, or V-ID terms respectively. For example, it is not possible to express equality between a class denotation and a value.

1. For example, 2 ≅ 3 is always false. In the past 2 := 3 caused "nice" errors in Fortran programs.

Data-terms can be used to describe objects, i.e. elements and classes, by using attribute values. In the following two data terms are given as example:

JAn[**age** : **32, boss** : **RSt [age** : **42], colleagues** :: **{SNe[], MKo[]},**
name : **"Jürgen Angele",**
subjects :: **{"Knowledge Acquisition", "Knowledge Engineering"}].**

MKo[**age** : **27, subjects** :: **{"Computer Theory"}].**[1]

2.1.3.4 The Difference Between Objects and Values

The difference between values and objects are illustrated by the following:

- A ***value*** cannot be further described by attributes.
- An ***element*** can be described by further attributes indicating features and relationships which hold true for it. A data term can be used to describe it using the values of allowed attributes. In the below data term, the value of the attribute "boss" is an E-ID term which indicates an object. Therefore this object which is the value of an attribute can also be described by an attribute. The following non-ground data-term
  X_E[age : X_V, boss : Y_E[age : X_V]]
  denotes all elements which have a boss with equal age, for example.
- A ***class*** can be described by further attributes which indicates features and relationships which hold for it. A data term can be used to describe it using the values of allowed attributes. In addition to elements, it can be used to model a set of real-world entities with common features. A class definition can be used to define allowed attributes for the elements of the class and subset relationships can be expressed using is-a terms. These definitions of signatures allow to define the well-typing condition (cf. [KLW90]).

2.1.3.5 The Well-typing Condition for Data Terms

If an attribute is applied to an element:

- The element must be an element of a class which defines or inherits this attribute for their elements.
- The value (or the values in the case of a set-valued attribute) of the attribute must be an element of the classes or value types or a subclass of the classes which are used to define the range restriction for the attribute.

If an attribute is applied to a class:

- The class must be a subclass of a class which defines this attribute or the attribute must be defined by its own class definition.

1. As you can see, in contrast to predicate logic not all attributes have to get a value.

- The value (or the values in the case of a set-valued attribute) of the attribute must be an element of the classes or value types or a subclass of the classes which are used to define the range restriction for the attribute.

2.1.3.6 Predicate Definitions and P-Terms

The use of predicates is similar to that in predicate logic. They are interpreted by relations over the domain. In addition, a predicate definition defines names and domains for the arguments. The following predicate definition “superior_of”

PREDICATE **superior_of**
subordinate : {**scientists**};
superior : {**professors**};
END

firstly improves the readability of the predicate “superior_of” by naming the arguments and thereby clarifying who is the superior and who is the subordinate. Secondly, the domains for the arguments are restricted, i.e. the second argument must be an element or a subset of the class “professors”.

The following given fact:

superior_of(subordinate: JAn[], superior: RSt[]).

indicates that “JAn” has “RSt” as superior. In extension to first-order logic, predicate terms need not to be flat.

superior_of(subordinate: JAn[], superior: RSt[age : 42]).

This fact implies in addition that the object denoted by “RSt” has the age “42”.

Π **= {superior_of(_,_)}**
A_{Π} **= {subordinate, superior}**

The ***well-typing condition*** for predicates:

> The i-th argument of a P-term must be an element of all classes or value types or a subclass of all classes which are used to define the domain restriction for the i-th argument in the according predicate definition.

2.1.3.7 Formulae

Elementary formula are F-, is-a, or P-terms. They are not atomic because the F-term:

JAn[boss : RSt [age : 42]].

is logically equivalent to the conjunction of the two F-terms:

JAn[boss : RSt[]] $\wedge$ **RSt[age : 42].**

Besides this, formulae are constructed in a way familiar from first-order logic.

A last example indicates how a set-valued attribute describing a class can be used:

$\forall x_E \forall x_V$
(scientists[all_subjects :: {x_V**}]** $\leftarrow$
x_E**[subjects :: {**x_V**}]** ε **scientists).**

The class “scientists” has as values for the attribute “all_subjects” all those that are subjects of

its elements. In the example this is:
{"Computer Theory", "Knowledge Acquisition", "Knowledge Engineering"}

2.1.3.8 Constraints

Constraints are formulae which describe necessary conditions. They check whether a given set of formulae violates integrity conditions. An example could be:

$\forall \mathbf{x_E} \neg (\mathbf{supervisor}(\mathbf{x_E}) \cong \mathbf{x_E})$.

It expresses, that no one can be its own supervisor.

2.1.3.9 The Graphical Representation of the Whole Example

Figure 7 presents the terminological (i.e. the class definitions) and the tables present factual knowledge (i.e. the is-element-of, equality, and data terms). Formulae are not represented graphically

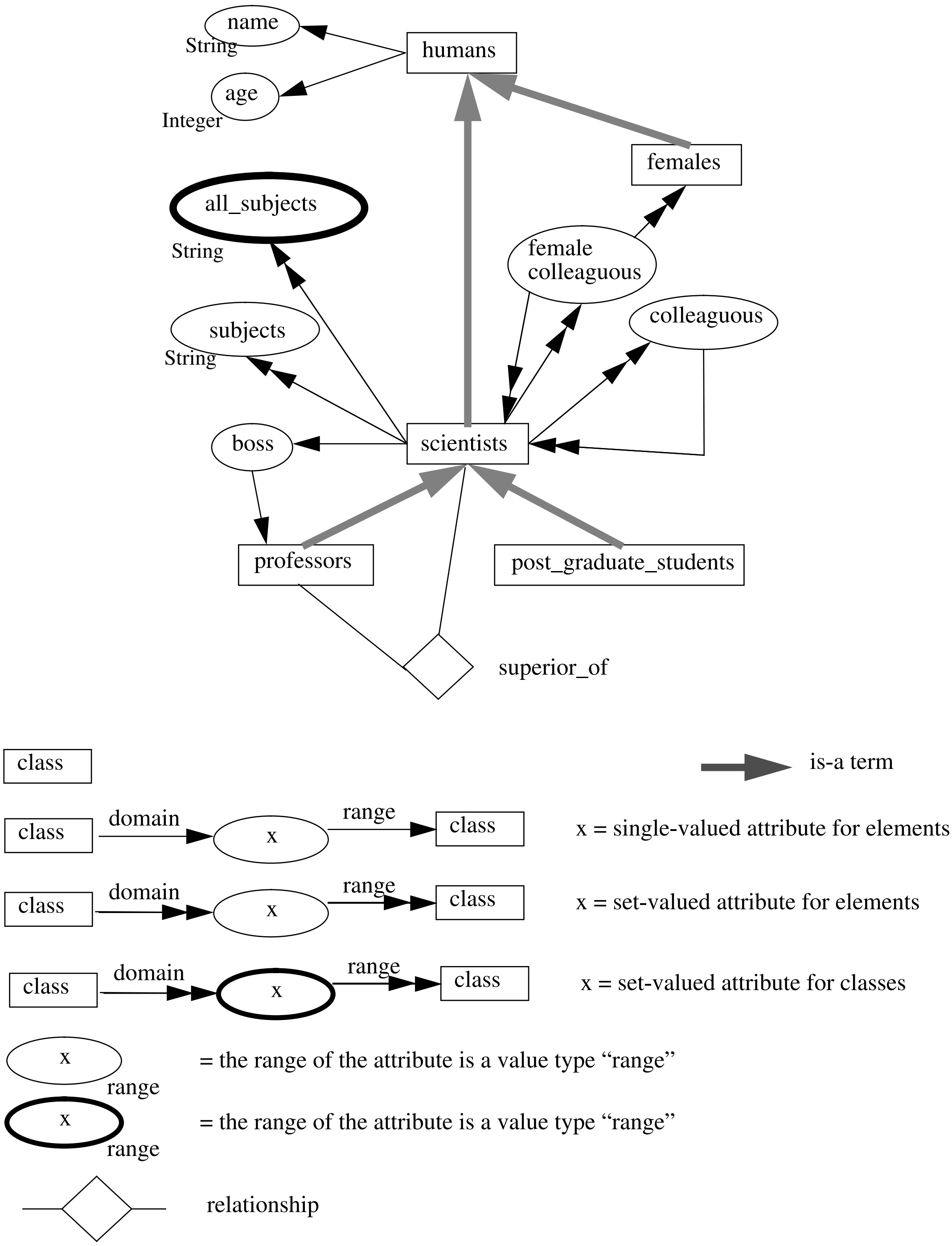


Figure 7. Graphical representation of the terminological knowledge.

class: humans

elements	age	name
JAn	32	“Jürgen Angele”
MKo	27	
SNe		
RSt = supervisor(JAn)	42	

class: females ≤ humans

elements	age	name

class: scientists ≤ {humans}

elements	age	boss	colleagues	female_ colleagues	name	subjects
JAn	32	RSt ≅ supervisor (JAn)	SNe, MKo		“Jürgen Angele”	“Knowledge Acquisition”, “Knowledge Engineering”
MKo	27					“Computer Theory”
SNe						
RSt = supervisor (JAn)	42					

class: professors ≤ {humans, scientists}

elements	age	boss	colleagues	female_ colleagues	name	subjects
RSt = supervisor (JAn)	42					

class: post_graduate_students ≤ {humans, scientists}

elements	age	boss	colleagues	female_ colleagues	name	subjects
JAn	32	RSt ≅ supervisor (JAn)	SNe, MKo		“Jürgen Angele”	“Knowledge Acquisition”, “Knowledge Engineering”
MKo	27					“Computer Theory”
SNe	27					

2.1.4 General Formal Semantics of L-KARL

In the following, a model-theoretical semantics is defined for a L-KARL language according to [KLW90]. First, a semantical structure is defined used to interpret the different elements of a L-KARL alphabet. Second, it is shown how these can be brought together to a formal semantics for attribute inheritance and well-typing. Third, variable assignment and truth values for formulae and definitions are defined. Last but not least, these can be summed up to define a ***model*** of a set of L-KARL formulae and definitions and how sets of L-KARL formulae and definitions ***entail*** other ones.

2.1.4.1 A Semantical Structure

Before defining a semantical structure for a L-KARL language some auxiliary definitions are introduced. Partial ordering, partial function, anti-monotonic function, and upwards-closed subsets are defined.

Def. 13 Definition of a partial ordering

- A partial ordering “$\leq_A$“ on a set “A” is a binary relation on “A” having the following properties:
  - “$\leq_A$“ is ***reflective***:
    $a \leq_A a$
    for all $a \in A$.

- “$\leq_A$“ is ***transitive***:
  $a \leq_A b$ and $b \leq_A c$ imply that $a \leq_A c$
  for all $a, b, c \in A$.
- “$\leq_A$“ is ***anti-symmetric***:
  $a \leq_A b$ and $b \leq_A a$ imply that $a = b$
  for all $a, b \in A$.

- “$\leq_A$” can be extended to form a partial ordering “$\leq_{A^k}$” ($0 < k$, k finite) on all tuples which are elements of A^k using the following extension:
  $(a_1,..., a_k) \leq_{A^k} (a'_1,..., a'_k)$
  if $a_i \leq_A a'_i$ for all $i = 1,..., k$.
- “$\leq_A$” can be extended to form a partial ordering “$\leq_{\wp(A)}$“ on all finite sets $t, t' \in \wp(A)$ using the following extension:
  $t \leq_{\wp(A)} t'$
  if $a \leq_{\wp(A)} a'$ for all $a \in t$, $a' \in t'$.
- “$\leq_A$” can be extended to form a partial ordering “$\leq_{\wp(A)^k}$” ($0 < k$, k finite) on all tuples which are element of the k-ary cartesian product of $\wp(A)$ using the following extension:
  $(t_1,..., t_k) \leq_{\wp(A)^k} (t'_1,..., t'_k)$
  if $t_i \leq_{\wp(A)^k} t'_i$ for all $i = 1,..., k$

Def. 14 Partial function

Let A and B denote arbitrary sets. A function f
$f: A \rightarrow B$
is called a ***partial function*** of A if it is only defined for a subset A´ of A.[1]

Def. 15 Anti-monotonic function

Let A and B denote arbitrary sets and “$\leq_A$“ a partial ordering on A and “$\leq_B$“ a partial ordering on B. A function f
$f: A \rightarrow B$
is called an ***anti-monotonic function*** vs. “$\leq_A$“ and “$\leq_B$“ if and only if:
the fact that $a' \leq_A a$ implies $f(a) \leq_B f(a')$.

Def. 16 Partial anti-monotonic function

Let A and B denote arbitrary sets and “$\leq_A$“ a partial ordering on A and “$\leq_B$“ a partial ordering on B. A partial function f on A
$f: A \rightarrow B$
is called a ***partial anti-monotonic function*** vs. “$\leq_A$“ and “$\leq_B$“ if and only if:
the fact that $a' \leq_A a$ and $f(a)$ is defined implies that

- $f(a')$ is defined and

1. $A \neq A'$ is not required, i.e. every function on A is also a partial function on A.

• $f(a) \leq_B f(a')$.

Def. 17 Upwardly-closed subsets

Let A denote an arbitrary set and "$\leq_A$" a partial ordering on A. A subset A´ of A is ***upwardly-closed*** vs. "$\leq_A$" if and only if:

the fact that $a \in A$ and $a' \leq_A a$ for an $a' \in A'$
imply that $a \in A'$.

$\wp(A)$ denotes the power set of a set A. $\wp_{\uparrow_{\leq_A}}(A)$ denotes the subset of $\wp(A)$ containing all upwardly-closed subsets of A vs. "$\leq_A$".

In the following, a semantical structure is defined.

Def. 18 Semantical structure of L-KARL

A ***semantical structure***, I, of a L-KARL language is a tuple

$I = \langle U_E, U_\Sigma, U_V, \leq_U, I_U, I_{E\rightarrow}, I_{\Sigma\rightarrow}, I_{E\rightarrow\rightarrow}, I_{\Sigma\rightarrow\rightarrow}, I_{E\Rightarrow}, I_{\Sigma\Rightarrow}, I_{E\Rightarrow\Rightarrow}, I_{\Sigma\Rightarrow\Rightarrow}, I_\Pi, I_{A\Pi}\rangle$

- U_E is a non empty set of all elements.
- U_Σ is a non empty set of all classes.
- $U_V = U_{VB} \cup U_{VI} \cup U_{VR} \cup U_{VS}$ is the union of the four sets:
  - U_{VB} = {False, True};
  - U_{VI} is the set of all integers;
  - U_{VR} is the set of all real numbers;
  - U_{VS} is the set of all strings of finite length.
- $U_V \cap U_E = U_V \cap U_\Sigma = U_E \cap U_\Sigma = \emptyset$.
- $U := U_E \cup U_\Sigma \cup U_V \cup \{U_{VB}, U_{VI}, U_{VR}, U_{VS}\}$.
- "$\leq_U$" is a partial ordering on U. The following conditions must hold for "$\leq_U$":
  - $x \leq_U y$ is equivalent to $y \in U_\Sigma \cup \{U_{VB}, U_{VI}, U_{VR}, U_{VS}\}$ or $x = y$.
  - $x \in U_{VB}$ is equivalent to $x \leq_U U_{VB}$.
  - $x \in U_{VI}$ is equivalent to $x \leq_U U_{VI}$.
  - $x \in U_{VR}$ is equivalent to $x \leq_U U_{VR}$.
  - $x \in U_{VS}$ is equivalent to $x \leq_U U_{VS}$.
- I_U interprets every element of $E \cup \Sigma \cup V$.[1] I_U is must be ***injective***[2] for Σ and V.

$$I_U(u) = \begin{cases} \text{is a total function } U_E^k \rightarrow U_E, & \text{if u is a k-ary (k>0) function symbol } u \in E \\ \text{is an element of } U_E, & \text{if u is a 0-ary constant } u \in E \\ \text{is an element of } U_\Sigma, & \text{if } u \in \Sigma \\ \text{is u}, & \text{if } u \in V. \end{cases}$$

• $I_{E\rightarrow}$ interprets each single-valued attribute $a \in A_{E\rightarrow}$ as a ***partial*** function[1]
$I_{E\rightarrow}(a): U_E \rightarrow U$.

• $I_{\Sigma\rightarrow}$ interprets each single-valued attribute $a \in A_{\Sigma\rightarrow}$ as a ***partial*** function
$I_{\rightarrow}(a): U_\Sigma \rightarrow U$.

• $I_{E\rightarrow\rightarrow}$ interprets each set-valued attribute $s \in A_{E\rightarrow\rightarrow}$ as a ***partial*** function
$I_{E\rightarrow\rightarrow}(s): U_E \rightarrow \wp(U)$.

• $I_{\Sigma\rightarrow\rightarrow}$ interprets each set-valued attribute $s \in A_{\Sigma\rightarrow\rightarrow}$ as a ***partial*** function
$I_{\Sigma\rightarrow\rightarrow}(s): U_\Sigma \rightarrow \wp(U)$.

• $I_{E\Rightarrow}$ interprets each single-valued attribute $a \in A_{E\rightarrow}$ as a ***partial anti-monotonic*** function in respect to "$\leq_U$" and "$\subseteq$"[2] having the ***upwardly-closed subsets*** of $\wp\uparrow_{\leq_U}(A)$ as its range with $A = U_\Sigma \cup \{U_{VB}, U_{VI}, U_{VR}, U_{VS}\}$.

$I_{E\Rightarrow}(a): U_E \cup U_\Sigma \rightarrow \wp\uparrow_{\leq_U}(U_\Sigma \cup \{U_{VB}, U_{VI}, U_{VR}, U_{VS}\})$.

• $I_{\Sigma\Rightarrow}$ interprets each single-valued attribute $a \in A_{\Sigma\rightarrow}$ as a ***partial anti-monotonic*** function in respect to "$\leq_U$" and "$\subseteq$" having the ***upwardly-closed subsets*** of $\wp\uparrow_{\leq_U}(A)$ as its range with $A = U_\Sigma \cup \{U_{VB}, U_{VI}, U_{VR}, U_{VS}\}$.

$I_{\Sigma\Rightarrow}(a): U_\Sigma \rightarrow \wp\uparrow_{\leq_U}(U_\Sigma \cup \{U_{VB}, U_{VI}, U_{VR}, U_{VS}\})$.

• $I_{E\Rightarrow\Rightarrow}$ interprets each set-valued attribute $s \in A_{E\rightarrow\rightarrow}$ as a ***partial anti-monotonic*** function in respect to "$\leq_U$" and "$\subseteq$" having the ***upwardly-closed subsets*** of $\wp\uparrow_{\leq_U}(A)$ as its range with $A = U_\Sigma \cup \{U_{VB}, U_{VI}, U_{VR}, U_{VS}\}$

$I_{E\Rightarrow\Rightarrow}(s): U_E \cup U_\Sigma \rightarrow \wp\uparrow_{\leq_U}(U_\Sigma \cup \{U_{VB}, U_{VI}, U_{VR}, U_{VS}\})$.

• $I_{\Sigma\Rightarrow\Rightarrow}$ interprets each set-valued attribute $s \in A_{\Sigma\rightarrow\rightarrow}$ as a ***partial anti-monotonic*** function in respect to "$\leq_U$" and "$\subseteq$" having the ***upwardly-closed subsets*** of $\wp\uparrow_{\leq_U}(A)$ as

---

1. Values are interpreted by themselves.

2. This restriction enforces the ***unique-name assumption*** for classes and values (cf. 2.1.4.5).

1. $I_{E\rightarrow}$, $I_{E\rightarrow\rightarrow}$, $I_{\Sigma\rightarrow}$, and $I_{\Sigma\rightarrow\rightarrow}$ interpret each attribute using a partial function because an attribute need not be defined for all elements or classes.

2. "$\leq_U$" is used as the partial ordering on U and "$\subseteq$" as the partial ordering on $\wp(U)$ for the definition of anti-monotony. In addition, "$\leq_U$" is used to define upward-closeness.
A partial function f on U is anti-monotonic if the facts that $v \leq_U u$ and f(u) is defined imply that
(1). f(v) is defined and
(2). $f(u) \subseteq f(v)$.
The first condition insures attribute inheritance, i.e. if a class is a subclass of the class or if an element is an element of the class it inherits the attributes. The second condition insures that the range restriction for an inherited attribute can be stronger for a subclass, i.e. 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 because it should be ensured 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´", cf. remark 5 at page 73.

its range with $A = U_\Sigma \cup \{U_{VB}, U_{VI}, U_{VR}, U_{VS}$

$$I_{\Sigma \Rightarrow\Rightarrow}(s): U_\Sigma \rightarrow \wp\uparrow_{\leq_U}(U_\Sigma \cup \{U_{VB}, U_{VI}, U_{VR}, U_{VS}\}) \ .$$

- I_Π interprets each k-ary predicate symbol $p \in \Pi$ by a relation $I_\Pi(p) \subseteq U^k$.
- $I_{A\Pi}$ interprets each k-ary predicate symbol $p \in \Pi$ as a k-ary tuple $(T_1,..., T_k)$ with

$$T_i \in \wp\uparrow_{\leq_U}(U_\Sigma \cup \{U_{VB}, U_{VI}, U_{VR}, U_{VS}\})$$

for all $i = 1,...,k$.

The set U_E is a set of objects which are used to interpret the element denotations of a L-KARL language. Similarly, the set U_Σ is a set of all classes which are used to interpret the class names and U_V is used to interpret each value. The union of these three sets and the set of names of the predefined types $\{U_{VB}, U_{VI}, U_{VR}, U_{VS}\}$ is the domain of discourse. It contains all elements, classes, values, and type names which are used to interpret the logical formulae.

"$\leq_U$" is a partial ordering on U which is used to interpret the is-a and is-element-of relationships. $u \leq_U c$ implies that

- "u" is an element of the class or value type "c" if "u" is an element and "c" is a class or value type or
- "u" is a subclass of the class or type "c" if "u" is a class and "c" is a class or value type;
- "c" is a class or a predefined type or u = c must hold true.

It is further necessary that

$$v \leq_U V$$

is equivalent to

$$v \in V$$

for every value type $V \in \{U_{VB}, U_{VI}, U_{VR}, U_{VS}\}$. This integrates the well-typing condition for the predefined types with the well-typing condition for classes as range restrictions.

The extension of "$\leq_U$" to tuples and sets according to Def. 13 "Definition of a partial ordering" on page 63 is required to define the well-typing conditions for predicates later on.

The function I_U interprets each element, class, and value denotation of a L-KARL language using a corresponding element of the domain of discourse. A function symbol is interpreted by a function and a constant is interpreted by an element of U. I_U is injective for class and value denotations because the unique-name assumption must hold true for them.

$I_{E\rightarrow}$ interprets each single-valued attribute $a \in A_{E\rightarrow}$ as a partial function

$$I_{E\rightarrow}(a): U_E \rightarrow U.$$

$I_{E\rightarrow}(a)$ is a ***partial*** function because attributes are not applicable to all elements. Only if the element belongs to a class for which the attribute is defined can the attribute be applied to the element. Because $I_{E\rightarrow}(a)$ is a function, it expresses the ***functional dependency*** between an element and its attribute value, i.e. the attribute value depends functionally on the element.

$I_{\Sigma\rightarrow}$ does the same for classes, i.e. not only elements, but also classes can be described by attributes.

$I_{E\rightarrow\rightarrow}$ interprets each set-valued attribute $s \in A_{E\rightarrow\rightarrow}$ as a partial function

$I_{E\rightarrow\rightarrow}(s)\colon U_E \rightarrow 2^U$.

$I_{E\rightarrow\rightarrow}(s)$ is a ***partial*** function because element attributes are not applicable to all elements. Only if the element belongs to a class for which the attribute is defined can the attribute be applied to the element. In contrast to $I_{E\rightarrow}(a)$, the value of such an attribute is a ***set***.

$I_{\Sigma\rightarrow\rightarrow}$ does the same thing for classes, i.e. not only elements but also classes can be described by set-valued attributes.

$I_{E\Rightarrow}$ interprets each single-valued attribute $a \in A_{E\rightarrow}$ for elements as a ***partial anti-monotonic*** function in respect to "$\leq_U$" and "$\subseteq$" having the ***upwardly-closed subsets*** of $\wp\uparrow_{\leq_U}(A)$ as its range with $A = U_\Sigma \cup \{U_{VB}, U_{VI}, U_{VR}, U_{VS}\}$.

$$I_{E\Rightarrow}(a)\colon U_E \cup U_\Sigma \rightarrow \wp\uparrow_{\leq_U}(U_\Sigma \cup \{U_{VB}, U_{VI}, U_{VR}, U_{VS}\}) \ .$$

This function is used to define the range restriction for single-valued attributes which are applied to elements. $I_{E\Rightarrow}$ maps each attribute to a ***partial*** (partial because an attribute is generally not used for all elements) function. This function returns for each element or class it is defined, the set of classes or value types which are used to define the range restriction for the attribute, if it applied to this element or to the elements of this class.

The range restriction can be different for different elements or classes. Because the value of this function is a set of classes and value types it is a mapping from the elements and classes on the powerset of all classes and value types.

In addition, this function has to be ***anti-monotonic***. This implies two conditions:

- If
  $I_{E\Rightarrow}(a)(v)$
  is defined and
  $u \leq_U v$
  holds true then
  $I_{E\Rightarrow}(a)(u)$
  must be defined. If $u \in U_\Sigma$, this means that "u" is a subclass of "v" and should therefore inherit all attributes from "v". If $u \in U_E$ this means that "u" is an element of "v" and "a" should therefore be defined for "u".

- If
  $I_{E\Rightarrow}(a)(v)$
  is defined and
  $u \leq_U v$
  holds true then
  $I_{E\Rightarrow}(a)(v) \subseteq I_{E\Rightarrow}(a)(u)$

must hold because a range restriction should be stronger for “u” then for “v”. If for example two classes c_2 and c_3 define the same attribute in their class definition and c_1 is a common subclass of both it inherits the range restrictions of both classes. The range restriction for the attribute applied to the elements of c_1 is therefore a superset of the range restrictions defined by c_2 and c_3.

$I_{E\Rightarrow}$ interprets each single-valued attribute for elements $a \in A_{E\rightarrow}$ as a function on $\wp_{\uparrow_{\leq_U}}(U_\Sigma \cup \{U_{VB}, U_{VI}, U_{VR}, U_{VS}\})$, i.e. on all ***upwardly-closed subsets*** of $U_\Sigma \cup \{U_{VB}, U_{VI}, U_{VR}, U_{VS}\}$. This ensures that every superclass $c' \in U_\Sigma$ of a class $c \in U_\Sigma$, i.e. $c \leq_U c'$, which is used to define the range restriction for the application of an attribute $a \in A_{E\rightarrow}$ to an element $u \in U_E$, i.e. $c \in I_{E\Rightarrow}(a)(u)$, is also an element of the definition of the range restriction, i.e. $c' \in I_{E\Rightarrow}(a)(u)$. If female $\leq_U$ humans and “female” is used as a range restriction, then it is also true that every attribute value must be an element of “humans”.[1]

$I_{E\Rightarrow\Rightarrow}$ does the same for set-valued attributes $s \in A_{E\rightarrow\rightarrow}$ for elements.

$I_{\Sigma\Rightarrow}$ interprets each single-valued attribute $a \in A_{\Sigma\rightarrow}$ for classes as a ***partial anti-monotonic*** function

$$I_{\Sigma\Rightarrow}(a): U_\Sigma \rightarrow \wp_{\uparrow_{\leq_U}}(U_\Sigma \cup \{U_{VB}, U_{VI}, U_{VR}, U_{VS}\}) \;.$$

The slight difference to $I_{E\Rightarrow}(a)$ is that $I_{\Sigma\Rightarrow}(a)$ is defined only for U_Σ. Because attributes for classes are not applicable to elements, these attributes need not be defined for elements. The converse does not hold true. The range restriction, even for attributes which are only applicable to elements, is defined by classes, i.e. by a class term.

$I_{\Sigma\Rightarrow\Rightarrow}$ does the same for set-valued attributes $s \in A_{\Sigma\rightarrow\rightarrow}$ for classes.

I_Π interprets each k-ary predicate symbol $p \in \Pi$ using a relation $I_\Pi(p) \subseteq U^k$ as is familiar from predicate logic.

$I_{A\Pi}$ interprets each k-ary predicate symbol $p \in \Pi$ as a k-ary tuple $(T_1,..., T_k)$ with

$$T_i \in \wp_{\uparrow_{\leq_U}}(U_\Sigma \cup \{U_{VB}, U_{VI}, U_{VR}, U_{VS}\}) \quad \text{for all } i = 1,..., k.$$

Each T_i defines the domain restriction for the i-th component of an element of the relation which interprets the corresponding predicate symbol. Each T_i is an upwardly-closed set for the same reason as is discussed in regard to $I_{E\Rightarrow}$.

---

1. Otherwise an interpretation could be defined where “c[[ELEMENT_ATT s :: {female}]]” is true and “c[[ELEMEN-T_ATT s :: {humans}]]” is false.

2.1.4.2 Formal Semantics for Attribute Inheritance and Well-Typing

Remark 3 Formal semantics for attribute inheritance and the inheritance of range restrictions.

The above given definitions of $I_{E\Rightarrow}$, $I_{E\Rightarrow\Rightarrow}$, $I_{\Sigma\Rightarrow}$, and $I_{\Sigma\Rightarrow\Rightarrow}$ define the ***formal semantics*** of attribute inheritance and the inheritance of range restrictions for attributes (cf. [KLW90]). In addition, the definition of $I_{A\prod}$ defines domain restrictions for predicates.

The following well-typing conditions ensure, that an interpretation does not violate the domain and range restrictions for attributes and the domain restrictions for predicates. An attribute can only be applied to an element or class if this is allowed by a corresponding class definition and the value of an attribute must be conform to all range restrictions of the attribute. Similarly, an element of the relation which interprets a predicate symbol must conform to the domain restriction of the predicate.

Def. 19 The Well-Typing Conditions

For all $a \in A_{E\rightarrow}$, $e \in U_E$:

- If $I_{E\rightarrow}(a)(e)$ is defined then so is $I_{E\Rightarrow}(a)(e)$;
- if $u = I_{E\rightarrow}(a)(e)$ for $u \in U$, then $u \leq_U c'$ for all $c' \in I_{E\Rightarrow}(a)(e)$.

For all $s \in A_{E\rightarrow\rightarrow}$, $e \in U_E$:

- If $I_{E\rightarrow\rightarrow}(s)(e)$ is defined then so is $I_{E\Rightarrow\Rightarrow}(s)(e)$;
- if $u \in I_{E\rightarrow\rightarrow}(s)(e)$ for $u \in U$, then $u \leq_U c'$ for all $c' \in I_{E\Rightarrow\Rightarrow}(s)(e)$.

For all $a \in A_{\Sigma\rightarrow}$, $c \in U_\Sigma$:

- If $I_{\Sigma\rightarrow}(a)(c)$ is defined then so is $I_{\Sigma\Rightarrow}(a)(c)$;
- if $u = I_{\Sigma\rightarrow}(a)(c)$ for $u \in U$, then $u \leq_U c'$ for all $c' \in I_{\Sigma\Rightarrow}(a)(c)$.

For all $s \in A_{\Sigma\rightarrow\rightarrow}$, $c \in U_\Sigma$:

- If $I_{\Sigma\rightarrow\rightarrow}(s)(c)$ is defined then so is $I_{\Sigma\Rightarrow\Rightarrow}(s)(c)$;
- if $u \in I_{\Sigma\rightarrow\rightarrow}(s)(c)$ for $u \in U$, then $u \leq_U c'$ for all $c' \in I_{\Sigma\Rightarrow\Rightarrow}(s)(c)$.

For all $p \in \prod$:

- If $(t_1,..., t_k) \in I_{\prod}(p)$ then $(\{t_1\},..., \{t_k\}) \leq_{\wp(A)^k} I_{A\prod}(p)$ if $I_{A\prod}(p)$ is defined.[1]

Remark 4 Formal semantics for well-typing.

The above given definitions ensure, that an interpretation does not violate the domain and range restrictions for attributes and the domain restrictions for predicates, i.e. it defines a ***formal semantics*** for these restrictions.

1. The use of predicates without defined argument restrictions, i.e. the use of P-terms without a corresponding predicate definition, is allowed.

2.1.4.3 Variable Assignment and Truth Values of Formulae

In the following, variable assignments are defined using the common way and extend it to form a term assignment in a modified manner because L-KARL provides more kinds of terms than predicate logic.

Def. 20 Variable and Term Assignment

- A ***variable-assignment***, ν, is a mapping $X_E \cup X_\Sigma \cup X_V \rightarrow U$.

$$\nu(x) = \begin{cases} u \in U_E, & \text{if } x \in X_E \\ u \in U_\Sigma, & \text{if } x \in X_\Sigma \\ u \in U_V, & \text{if } x \in X_V \end{cases}$$

In the following, this variable assignment is extended to form a ***term assignment.***

- It is extended to E-ID terms f(..., T,...) in the following way.
  - $\nu(f(..., T,...)) = I_U(f)\ (..., \nu(T),...)$, , if the arity of f is $k \neq 0$.
  - $\nu(f) = I_U(f)$, if the arity of f is $k = 0$.
- For a ground C-ID term “c”, $\nu(c) = I_U(c)$.[1]
- For a ground value term “v”, $\nu(v) = I_U(v)$.[2]
- For a data-term $T[a_1 : T_1,..., a_r : T_r, s_1 :: \{S_{11},..., S_{1m_1}\},..., s_t :: \{S_{t1},..., S_{tm_t}\}]$

$$\nu(T[a_1 : T_1,..., a_r : T_r, s_1 :: \{S_{11},..., S_{1m_1}\},..., s_t :: \{S_{t1},..., S_{tm_t}\}]) = \nu(T).$$[3]

The truth value for class terms, is-a terms, class definitions, predicate definitions, F-terms, and P-terms is defined in the following.

A class term is satisfied, i.e. proved true, by an interpretation if

- its class name is interpreted by an element of U_Σ;
- if it is true for each attribute which is defined in it that the function which interprets the range restriction of the attribute maps the class on a superset of the range restriction which is defined by the class term. It must be a superset for two reasons. First, the class can inherit additional range restrictions for the attribute from superclasses which are not defined in its own class terms. Second, several class terms can exist for the same class.

Def. 21 Satisfaction of a class term

A class term T with
T := c[[
CLASS_ATT
$ca_1 : \{ca_{11},..., ca_{1k_1}\}$;

1. Non-ground class terms are variables and are therefore covered by the variable assignment.
2. Non-ground value terms are variables and are therefore covered by the variable assignment.
3. Cf. the definition of the satisfaction of a data-term in Def. 25 “Satisfaction of an F-term” on page 74.

$$
\begin{array}{l}
\qquad ...; \\
\qquad ca_r : \{ca_{r1},..., ca_{rk_r}\}; \\
\qquad cs_1 :: \{cs_{11},..., cs_{1l_1}\}; \\
\qquad ...; \\
\qquad cs_t :: \{cs_{t1},..., cs_{tl_t}\}; \\
\text{ELEMENT\_ATT} \\
\qquad ea_1 : \{ea_{11},..., ea_{1m_1}\}; \\
\qquad ...; \\
\qquad ea_u : \{ea_{u1},..., ea_{rm_u}\}; \\
\qquad es_1 :: \{es_{11},..., es_{1n_1}\}; \\
\qquad ...; \\
\qquad es_v :: \{es_{v1},..., es_{vn_v}\}; \\
]]
\end{array}
$$

is satisfied by a semantical structure I, denoted $I \models T$, if and only if the following conditions hold:

- $I_U(c) \in U_\Sigma$ and $I_{\Sigma\to}(exists)(I_U(c))$ is defined.[1]
- For each single-valued class attribute definition $ca_i : \{ca_{i1},..., ca_{ik_i}\}$, i = 1,..., r:
  - $I_{\Sigma\Rightarrow}(ca_i)(I_U(c))$ is defined;
  - $\{I_U(ca_{i1}),..., I_U(ca_{ik_i})\} \subseteq I_{\Sigma\Rightarrow}(ca_i)(I_U(c))$.
- For each set-valued class attribute definition $cs_j :: \{cs_{j1},..., cs_{jl_j}\}$, j = 1,..., t:
  - $I_{\Sigma\Rightarrow\Rightarrow}(cs_j)(I_U(c))$ is defined;
  - $\{I_U(cs_{j1}),..., I_U(cs_{jl_j})\} \subseteq I_{\Sigma\Rightarrow\Rightarrow}(cs_j)(I_U(c))$.
- For each single-valued element attribute definition $ea_i : \{ea_{i1},..., ea_{im_i}\}$, i = 1,..., u:
  - $I_{E\Rightarrow}(ea_i)(I_U(c))$ is defined;
  - $\{I_U(ea_{i1}),..., I_U(ea_{im_i})\} \subseteq I_{E\Rightarrow}(ea_i)(I_U(c))$.
- For each set-valued element attribute definition $es_j :: \{es_{j1},..., es_{jn_j}\}$, j = 1,..., v:
  - $I_{E\Rightarrow\Rightarrow}(es_j)(I_U(c))$ is defined;
  - $\{I_U(es_{j1}),..., I_U(es_{jn_j})\} \subseteq I_{E\Rightarrow\Rightarrow}(es_j)(I_U(c))$.

An is-a term $c \leq c'$ of two C-ID terms is satisfied, i.e. proved true, by an interpretation and a term assignment if the term assignment maps “c” and “c´” on a pair of classes for which “$\leq_U$“ holds true.

Def. 22 Satisfaction of an is-a term

An ***is-a term*** T

$$T := c \leq c',$$

1. Cf. the footnote to Def. 3 “Class term” on page 49.

is satisfied by a semantical structure I and a term assignment ν, denoted I $\models_\nu$ T, if and only if the following conditions hold:

- $\nu(c), \nu(c') \in U_\Sigma$
- $\nu(c) \leq_U \nu(c')$.

Remark 5 Upwardly-closed subsets and truth-value of class terms.

If c[CLASS_ATT ca : {ca_1};] and $ca_1 \leq ca_2$ are true, then c[CLASS_ATT ca : {ca_2};] is true. This is a consequence of the upwardly-closedness of the sets defining the range restrictions for attributes. The same holds if ca is a set-valued class attribute or a single-valued or set-valued element attribute.

A class definition is satisfied, i.e. proved true, by an interpretation if its class term and all its is-a terms are evaluated to be true.

Def. 23 Satisfaction of a class definition

An ***class definition*** T

$T := c[[...]] \wedge c \leq c_1 \wedge ... \wedge c \leq c_n$

is satisfied by a semantical structure I, denoted I $\models$ T, if and only if the following conditions hold true:

- the class term c[[...]] is satisfied by the semantical structure I
- and every ground is-a term[1] $c \leq c_i$ for i = 1,...,n is satisfied by the semantical structure I.

A predicate definition is satisfied, i.e. proved true, by an interpretation if

- its predicate name is interpreted by an element of I_Π;
- the interpretations of the class names and value types used to define the domain restriction of each argument are a subset of the corresponding domain restrictions for the predicate symbol and the argument as it is interpreted by $I_{A\Pi}$.

Def. 24 Satisfaction of a predicate definition

A ***predicate definition*** P

```
P :=  PREDICATE p
          p1 : { c11 ,..., c1r1 };
          ...,
          pk : { ck1 ,..., ckrk };
      END
```

1. Class definitions contain only ground is-a terms.

is satisfied by a semantical structure I denoted I |= P, if and only if the following conditions hold

- $I_{\prod}(p)$ is defined.
- $I_{A\prod}(p)$ is defined.
- $\{I_U(c_{i1}),..., I_U(c_{ir_i})\} \subseteq T_i$, i = 1,...,k, with $(T_1, ..., T_k) := I_{A\prod}(p)$

Further on, the satisfaction of F-terms, i.e. of is-element-of terms, equality terms, and data terms is defined. An is-element-of term is satisfied if the corresponding "$\leq_U$" relationship holds true for the term assignment of its ID terms. An equality term is satisfied if the two ID terms are mapped to the same object or value by the term assignment (cf. section 2.1.4.5). A data term is satisfied

- if the functions which interpret its attributes are defined for the object which interprets the ID term of the data term according to the term assignment;
- if the interpretation of the value for every single-valued attribute according to the term assignment is identical to the value of the function which interprets the attribute applied to the term assignment of the ID term of the data term;
- if the set of all interpretations of the values for every set-valued attribute according to the term assignment is a subset of the set return by the function which interprets the attribute applied to the term assignment of the ID term of the data term;
- and if every data term contained by the data term in question is true.

Def. 25 Satisfaction of an F-term

- An ***is-element-of term*** T
  $T := u \, \varepsilon \, c$,
  is satisfied by a semantical structure I and a term assignment ν, denoted $I \models_\nu T$, if and only if the following given conditions hold true:
  - $\nu(u) \leq_U \nu(c)$,
  - $\nu(u) \in U_E$,
  - $\nu(c) \in U_\Sigma$.
- An ***equality term*** T
  $T := o \cong o'$
  is satisfied by a semantical structure I and a term assignment ν, denoted $I \models_\nu T$, if and only if the following given condition hold true:
  - $\nu(o) = \nu(o') \in U$
- A ***data term*** T
  $T := o[a_1 : T_1,..., a_r : T_r, s_1 :: \{S_{11},..., S_{1m_1}\},..., s_t :: \{S_{t1},..., S_{tm_t}\}]$,
  is satisfied by a semantical structure I and a term assignment ν, denoted $I \models_\nu T$, if and only if the following conditions hold true:
  - $\nu(o) \in U_E$ and

- $I_{E\rightarrow}(a_i)(\nu(o))$ is defined for all i = 1,..., r;
- $\nu(T_i) = I_{E\rightarrow}(a_i)(\nu(o))$ for all i = 1,..., r;
- $I \models_\nu T_i$ for all i = 1,..., r;
- $I_{E\rightarrow\rightarrow}(s_j)(\nu(o))$ is defined for all j = 1,..., t;
- $\{\nu(S_{j1}),..., \nu(S_{jm_j})\} \subseteq I_{E\rightarrow\rightarrow}(s_j)(\nu(o))$ for all j = 1,..., t;
- $I \models_\nu S_{ji}$ for all j = 1,..., t and i = 1,..., m_j.

- or $\nu(o) \in U_\Sigma$ and
  - $I_{\Sigma\rightarrow}(a_i)(\nu(o))$ is defined for all i = 1,..., r;
  - $\nu(T_i) = I_{\Sigma\rightarrow}(a_i)(\nu(o))$ for all i = 1,..., r;
  - $I \models_\nu T_i$ for all i = 1,..., r;
  - $I_{\Sigma\rightarrow\rightarrow}(s_j)(\nu(o))$ is defined for all j = 1,..., t;
  - $\{\nu(S_{j1}),..., \nu(S_{jm_j})\} \subseteq I_{\Sigma\rightarrow\rightarrow}(s_j)(\nu(o))$ for all j = 1,..., t;
  - $I \models_\nu S_{ji}$ for all j = 1,..., t and i = 1,..., m_j.

A predicate term is satisfied,

- if the interpretation of the predicate symbol is defined;
- if the tuple of the interpretation of the arguments is an element of the relation with which the predicate symbol is interpreted;
- if every argument, i.e. the corresponding data term, is satisfied.

Def. 26 Satisfaction of a P-term

A P-term P

$$P = p(p_1{:}T_1,..., p_k{:}T_k)$$

is satisfied by a semantical structure I and a term assignment ν, denoted $I \models_\nu P$, if and only if the following conditions hold true:

- $I_\Pi(p)$ is defined;
- $(\nu(T_1),..., \nu(T_k)) \in I_\Pi(p)$;
- $I \models_\nu T_i$ for all i=1,..., k.[1]

The truth value of definitions and elementary formulae is defined above. The truth value of composite formulae is defined as in predicate logic.

- The conjunction of two formulae is true if both formulae are true.
- The disjunction of two formulae is true if at least one of the formulae is true.

1. The names p_i of the arguments do not receive a formal semantics. They are only used to improve readability.

- The negation of a formula is true if the formula is not true.
- The implication of two formulae is true if the equivalent disjunction is true.
- An existential-quantified formula is true if there exists at least one term assignment for which the formula is true.
- An all-quantified formula is true if the formula is true for every existing term assignment.

Def. 27 Satisfaction of a composed formula

For formulae built up by other formulae and logical connectives, the satisfaction is defined by mutual induction. Let ψ, ϕ be two formulae of a L-KARL language, I a semantical structure and ν a term assignment:

- $\psi \wedge \phi$ is satisfied by the semantical structure I and the term assignment ν, written by
  $I \models_\nu \psi \wedge \phi$
  if ψ and ϕ are satisfied by the semantical structure I and the term assignment ν, i.e.
  $I \models_\nu \psi$ and $I \models_\nu \phi$.
- $\psi \vee \phi$ is satisfied by the semantical structure I and the term assignment ν, written by
  $I \models_\nu \psi \vee \phi$
  if ψ or ϕ (or both) are satisfied by the semantical structure I and the term assignment ν, i.e.
  $I \models_\nu \psi$ or $I \models_\nu \phi$.
- $\neg\psi$ is satisfied by the semantical structure I and the term assignment ν, written by
  $I \models_\nu \neg\psi$
  if ψ is **not** satisfied by the semantical structure I and the term assignment ν, i.e.
  $I \not\models_\nu \psi$.
- $\psi \leftarrow \phi$ is satisfied by the semantical structure I and the term assignment ν, written by
  $I \models_\nu \psi \leftarrow \phi$
  if $\psi \vee \neg\phi$ are satisfied by the semantical structure I and the term assignment ν, i.e.
  $I \models_\nu \psi \vee \neg\phi$.
- If ψ is a quantified formula $\psi = (\forall X \phi)$ then
  $I \models_\nu \psi$
  if for ***every*** term assignment μ which agrees with ν every where except perhaps X,
  $I \models_\mu \phi$.
- If ψ is a quantified formula $\psi = (\exists X \phi)$ then
  $I \models_\nu \psi$
  if for ***some*** term assignment μ which agrees with ν every where except perhaps X,
  $I \models_\mu \phi$.

Def. 28 Closed formula

If a formula ϕ is ***closed***, i.e. it does not contain any free variables, its meaning is independent of a term assignment, and

$I \models \phi$

if ϕ is satisfied by I.

Remark 6 Ground formula and closed formulae

The satisfaction of class terms, class definitions, and predicate definitions was independently defined from the variable assignment because these terms are ground. Every ground formulae is also a closed one.

Only closed formulae are considered in the following.

2.1.4.4 Models and Entailment

In the following, untyped semantical structures, models, entailment, and logical equivalence are defined. ***Untyped semantical structures*** are used in the next section in order to define whether or not a minimal model of a set of L-KARL definitions and formulae exists. The definition of a ***model*** is one of the key points in this chapter because a model defines a possible meaning of a set of L-KARL expressions. The definition of ***entailment*** answers the question which L-KARL expressions can be concluded by a given set of L-KARL expressions. The definition of ***locigal equivalence*** will be used during the following.

Def. 29 Untyped semantical structure

An untyped semantical structure I´ of a L-KARL language L is a semantical structure of L which need not fulfil the well-typing conditions. It is written

$I' \models_u \phi$

if ϕ is a closed formula or definition satisfied by the untyped semantical structure I´.

It is easy to see that every typed semantical structure is also an untyped one.

Def. 30 Model

A semantical structure I is a ***model*** of a formula or a definition ϕ if $I \models \phi$.

A semantical structure I is a model of a set of formulae and definitions S if

$I \models \phi$ for all $\phi \in S$.

Def. 31 Untyped model

An untyped semantical structure I´ is an ***untyped model*** of a formula or a definition ϕ if

$I' \models_u \phi$.

An untyped semantical structure I´ is an untyped model of a set of formulae and definitions S if

$I´ \models_u \phi$ for all $\phi \in S$.

Def. 32 Entailment

A set of formulae and definitions S ***entails*** a formula or definition ϕ, written as

$S \models \phi$

if every model of S is also a model of ϕ.

A set of formulae and definitions S ***entails without regarding well-typing*** a formula or definition ϕ, written as

$S \models_u \phi$

if every untyped model of S is also an untyped model of ϕ.

Def. 33 Logical equivalence

Two formulae ϕ and ψ are logically equivalent if and only if

$\{\phi\} \models \psi$ and $\{\psi\} \models \phi$.

It is written as $\phi \equiv \psi$.

2.1.4.5 Equality Terms, Functional Attributes (i.e., Single-Values Attributes), and the Unique-Name Assumption

Let the following set of formulae and definitions S over an alphabet ({o,o´,o´´}, {c,c´,c´´}, Boolean, Integer, Real, String,{a}, ∅, ∅, ∅, ∅, ∅, ∅, ∅, ∅) of a L-KARL language be given:

S = {o[a:o´], o[a:o´´], c[[ELEMENT_ATT a:{};]], o ε c}

Then the equality term

o´ ≅ o´´

can be logically entailed from S, i.e.

S |= o´ ≅ o´´.[1]

This follows from the functionality of the attribute “a”. A semantical structure I which is to be a model of S must therefore interpret o´ and o´´ using the same element, i.e.

$I_U(o´) = I_U(o´´)$.

Similarly, if S´ = {o´ ≅ o´´} is given. A semantical structure I which should be a model of S´ must interpret o´ and o´´ by the same element, i.e.

$I_U(o´) = I_U(o´´)$.

The situation is different for values and classes because L-KARL assumes the ***unique-name assumption*** for class and value denotations. Let the following set of formulae and definitions S´´ be given:

S´´ = {o[a:c´], o[a:c´´], c[[ELEMENT_ATT a:{};]], o ε c}

1. It is easy to show that every model of S is also a model of o´ ≅ o´´.

Then the equality term

$c' \cong c''$

can be logically entailed from S'', i.e.

$S'' \models c' \cong c''$.

This follows from the functionality of the attribute "a". A semantical structure I which is to be a model of S'' must therefore interpret c' and c'' using the same class, i.e.

$I_U(c') = I_U(c'')$.

But this is not possible, because I_U is injective for Σ. Therefore, S'' has no model. The same result holds for the formulae set $\{c' \cong c''\}$.

Remark 7 Implicit equalities

As discussed above, data terms using single-valued (functional) attributes can define implicit equalities between ID terms. Therefore, additional equality terms can be entailed from them. In the case of different ground C-ID or V-ID terms this leads to a contradiction, i.e. the formulae set has no model. In the case of ground E-ID terms a semantical structure which should be a model of the formulae set must interpret these E-ID terms using the same element.

Remark 8 Equalities between ID terms of different types

An equality term $o \cong o'$ where "o" and "o′" are not **both** E-ID terms, C-ID terms, or V-ID terms is never satisfied by a semantical structure. A model of it would assume:

$\nu(o) = \nu(o')$

but this is impossible because of

$U_V \cap U_E = U_V \cap U_\Sigma = U_E \cap U_\Sigma = \emptyset$.

Therefore, a set of data terms which assumes implicit equalities between ID terms of different types never has a model.

2.1.5 Herbrand semantics of L-KARL

2.1.5.1 Herbrand Interpretations

In the following ***Herbrand models*** are introduced as semantics for L-KARL. This is a standard way of defining models for a set of formulae (cf. [Llo87] for details and broader explanations). Formulae are restricted to clauses because the following equivalence holds true for clauses:

φ has a model $\Leftrightarrow$ φ has a Herbrand model.

The formulae must be clauses if the equivalence should hold true. In [Llo87] the following example is given for first-order logic:

$\{p(a), \exists x\, \neg p(x)\}$

which has several models but no Herbrand model. The interpretation with the domain {a,b} which interprets "p" using the relation {(a)} is a model of the formulae set. The Herbrand base is HB = {a}. Therefore ∅ and {p(a)} are the only two Herbrand interpretations. Both are no models of the formula set.

As already discussed, data terms may be embedded in other data terms up to an arbitrary depth. To simplify the proof theory, we will restrict our attention to clauses consisting of ***flat*** terms only. This does not limit the generality because any predicate or a data term can be transformed into an equivalent conjunction of flat terms. For example:

$o_1[a : o_2[b : T]] \equiv o_1[a : o_2] \wedge o_2[b : T]$

Def. 34 flat

- Is-element-of terms, is-a terms, equality terms, class terms, class definitions, and predicate definitions are flat.
- A term T is singular if it is a value term T or a data term T'[].
- A data term
  $o[..., a_i : T_i,..., s_j :: \{S_{j1},..., S_{jm_j}\},...]$
  is flat if all T_i and S_i are singular.
- A predicate term
  $p(T_1,..., T_k)$
  is flat if all T_i are singular.

The following definition introduces the concepts ***literal*** and ***clause***, which are the only restricted forms of formulae which are considered further on.

Def. 35 Literal and clause

A ***positive literal*** is a flat is-a, F-, or P-term and a ***negative literal*** is the negation of a flat is-a, F-, or P-term. A ***literal*** is a positive or negative literal.

A ***clause*** is a formula

$\forall x_1 ... \forall x_s (l_1 \vee ... \vee l_m)$.

where each $l_1,..., l_m$ is a literal and $x_1,..., x_s$ are all variables occurring in $l_1 \vee ... \vee l_m$.

Now, ***Herbrand universe (HU), Herbrand base (HB), and Herbrand interpretation (HI)*** are defined.

Def. 36 Herbrand universe, HU

For a L-KARL language L with the alphabet (E, Σ, Boolean, Integer, Real, String, $A_{E\rightarrow}$, $A_{\Sigma\rightarrow}$, $A_{E\rightarrow\rightarrow}$, $A_{\Sigma\rightarrow\rightarrow}$, Π, A_Π, X_E, X_Σ, X_V), the ***Herbrand universe*** HU of L is ID* = E* ∪ Σ ∪ V, i.e. the set of all ground E-ID terms, ground C-ID terms, and ground V-ID terms.

A Herbrand Universe determines the set of all constants which can be used in terms. It can be used to define the set of all ground elementary formulae and definitions which can be constructed by these terms.

Def. 37 Herbrand base, HB

For a L-KARL language L with the alphabet (E, Σ, Boolean, Integer, Real, String, $A_{E\rightarrow}$, $A_{\Sigma\rightarrow}$, $A_{E\rightarrow\rightarrow}$, $A_{\Sigma\rightarrow\rightarrow}$, $\prod$, $A_{\prod}$, X_E, X_Σ, X_V) the ***Herbrand base*** HB is the set of all ground elementary formulae (cf. Def. 9 "Formulae" on page 52) and definitions[1].

The next definition ensures that the ground equality terms of a Herbrand interpretation do not assume the identity of ID terms of different types or the identity of different elements of Σ or V, i.e. they do not assume an ***equality error***.

Def. 38 Equality error

A set of ground equality terms has an ***equality error*** if and only if it contains an equality term $o \cong o'$ with

$o \neq o'$ and $o \notin E^*$ or $o' \notin E^*$.

Def. 39 Herbrand interpretation, HI

Let HI $\subseteq$ HB, HB denotes the Herbrand base of a L-KARL language L. HI is a ***Herbrand interpretation*** of a L-KARL language L if and only if

- it is closed under the logical entailment ($\models$)[2], i.e. if $\phi \in$ L is a ground elementary formulae or a definition and
  HI $\models \phi$ then $\phi \in$ HI, and
- the subset of equality terms of HI does not have an equality error.

Subsets of a Herbrand base HB can be used to define an ***interpretation*** for a L-KARL language L. All ground is-a terms, F-terms, P-terms and definitions which are elements of HI are considered as true. Element, class and value denotations which do not have a truth value but denote some entity, are interpreted by themselves. Later on the way will be discussed in which the truth value of composed formulae and non-ground formulae is defined.

But **not** every subset of a Herbrand base HB will be considered as an interpretation, i.e. as a Herbrand interpretation. Only those subsets of HB which are closed under logical entailment are considered as Herbrand interpretations. In predicate logic, every subset of a Herbrand base is closed under logical entailment. Therefore, this must not be enforced explicitly. In L-KARL the situation is different. The data-term

$T_1 := o[a :: \{b,c\}]$

1. Definitions are always ground.
2. Cf. Def. 32 "Entailment" on page 78.

entails the data-terms

$T_2 := o[a :: \{b\}]$

$T_3 := o[a :: \{c\}]$

because every model of T_1 is also a model of T_2 and T_3. If closeness is explicitly enforced for logical entailment, then the same situation is reached as for predicate logic:

$HI \models \phi \Leftrightarrow \phi \in HI$

Therefore, the truth value of a closed elementary formula is true for a Herbrand interpretation HI if and only if it is an element of the Herbrand interpretation HI. In addition, a Herbrand interpretation must not contain an equality error, i.e. it should not violate the unique-name assumption for classes and values or assume the identity of ID terms of different type.

Predicate logic with equality also requires logical closeness by ensuring that the equality predicate defines a ***congruence relation*** (cf. [SpA91], [Sch92b]):

- symmetry: $\models a \cong a$;
- reflexivity: $a \cong b \models b \cong a$;
- transitivity: $a \cong b$ and $b \cong c \models a \cong c$;
- congruence:
  - $a \cong b \models f(a_1,..., a_{i-1}, a, a_{i+1},..., a_n) \cong f(a_1,..., a_{i-1},b, a_{i+1},..., a_n)$ for every n-ary function symbol "f";
  - $a \cong b$ and $p(a_1,..., a_{i-1}, a, a_{i+1},..., a_n) \models p(a_1,..., a_{i-1},b, a_{i+1},..., a_n)$ for every n-ary predicate symbol "p".

A semantical structure of a L-KARL language interprets the equality terms by the equality on the universe, i.e. $a \cong b$ implies that $I_U(a) = I_U(b)$, whereas Herbrand interpretations of a L-KARL language weak "≅" to a congruence relationship. This difference is discussed in section 2.1.5.3.[1]

As untyped semantical structures were considered, i.e. semantical structures for which the well-typing conditions do not hold, untyped Herbrand interpretations are regarded. The closeness condition is relaxed to closed under "$\models_u$".

Def. 40 Untyped Herbrand Interpretation, HI´

Let HI' $\subseteq$ HB and HB denotes the Herbrand base of a L-KARL language L; HI´ is an ***untyped Herbrand interpretation*** of a L-KARL language L if and only if

- it is closed under the logical entailment "$\models_u$", i.e. if $\phi \in L$ is a ground elementary formulae or definition and
  $HI' \models_u \phi$ then $\phi \in HI'$, and
- the subset of equality terms of HI does not have an equality error.

1. "Though with respect to their models these two approaches to equality are different (the second variant admits more models than the first one), they will be shown to be equivalent from a logical point of view. " [SpA91], p. 131.

Remark 9 The following hold true for a (well-typed) Herbrand interpretations HI:

- u[a : T[]] ∈ HI implies that the attribute must be defined for u:
  - if u ∈ E* then c[[ELEMENT_ATT a : {};]] ∈ HI and u ε c ∈ HI for a c ∈ ∑.
  - if u ∈ ∑ then u[[CLASS_ATT a : {};]] ∈ HI.
- u[s :: {S[]}] ∈ HI implies that the attribute must be defined for u:
  - if u ∈ E* then c[[ELEMENT_ATT s :: {};]] ∈ HI and u ε c ∈ HI for a c ∈ ∑.
  - if u ∈ ∑ then u[[CLASS_ATT s :: {};]] ∈ HI.
- u[a : T[]], c[[ELEMENT_ATT a : {c'};]] ∈ HI and u ε c ∈ HI imply that the range restriction of the element attribute must hold true:
  - T ≤ c' ∈ HI or
  - T ε c' ∈ HI.
- u[s :: {S[]}], c[[ELEMENT_ATT s :: {c'};]] ∈ HI and u ε c ∈ HI imply that the range restriction of the element attribute must hold true:
  - S ≤ c' ∈ HI or
  - S ε c' ∈ HI.
- u[a : T[]] ∈ HI and u[[CLASS_ATT a : {c};]] ∈ HI imply that the range restriction of the class attribute must hold true:
  - T ≤ c ∈ HI or
  - T ε c ∈ HI.
- u[s :: {S[]}] ∈ HI and u[[CLASS_ATT s :: {c};]] ∈ HI imply that the range restriction of the class attribute must hold true:
  - S ≤ c ∈ HI or
  - S ε c ∈ HI.
- p(T_1[],..., T_k[]) ∈ HI, PREDICATE p
  p_1 : {c_1};
  ...
  p_k : {c_k};
  END ∈ HI

  imply that the domain restriction of the predicate must hold true:
  - $T_i \leq c_i$ ∈ HI or
  - T_i ε c_i ∈ HI

  for all i = 1,..., k.

These follow from the well-typing conditions (cf. Def. 19 “The Well-Typing Conditions” on page 70). Therefore for an ***untyped*** Herbrand interpretation HI´ they need not hold true.

2.1.5.2 Herbrand Models

In the last section Herbrand universe, Herbrand base and Herbrand interpretation were defined. It was shown how a Herbrand interpretation can be used to define truth values for definitions and ground elementary formulae. This section defines truth values for formulae, i.e. for clauses, which need not be ground or elementary. For this purpose the auxiliary definitions ***substitution*** and ***instance*** were introduced. This allows the general definition of ***truth values for clauses according to a Herbrand Interpretation***. The definition of a ***Herbrand model*** sums up these definitions.

Def. 41 Substitution and ground substitution

A ***substitution*** Θ is a finite set of the form $\{v_1/t_1,..., v_n/t_n\}$, where each v_i is a variable, i.e. it is an element of $X_E \cup X_\Sigma \cup X_V$, each t_i is an ID term distinct from v_i and the variables $v_1,..., v_n$ are distinct. In addition, the following conditions must hold:

- if $v_i \in X_E$ then t_i is an E-ID term;
- if $v_i \in X_\Sigma$ then t_i is a C-ID term;
- if $v_i \in X_V$ then t_i is a V-ID term.

A substitution Θ is called a ***ground substitution*** if the t_i are all ground ID terms.

Def. 42 Instance and ground instance

Let $\Theta = \{v_1/t_1,..., v_n/t_n\}$ be a substitution and Λ a clause.

- Then $\Lambda\Theta$, the ***instance*** of Λ by Θ, is the clause obtained from Λ by simultaneously replacing each occurrence of the variable v_i in Λ by the term t_i, i=1,...,n.
- If $\Lambda\Theta$ is ground, then $\Lambda\Theta$ is called a ***ground instance*** of Λ.

Def. 43 Truth values for literals and clauses according to Herbrand interpretations

Let HI be a Herbrand interpretation to a L-KARL language L. ***Truth in Herbrand interpretations HI*** is determined by the following rules:

- A ground positive literal l is ***true*** according to HI if and only if $l \in$ HI.
- A ground negative literal $\neg l$ is ***true*** according to HI if and only if $l \notin$ HI.
- A ground clause
  $l_1 \vee ... \vee l_n$.
  is ***true*** according to HI if and only if at least one literal l_i, i=1,...,n, is true according to HI.
- A clause
  $\Lambda = \forall x_1 ... \forall x_s (l_1 \vee ... \vee l_m)$.
  is ***true*** according to HI if and only if all ground instances of Λ are true according to HI.

Def. 44 Herbrand model and untyped Herbrand model

• Let S be a set of clauses and definitions and H a Herbrand interpretation of it. If every clause and definition in S is true in H then H is a ***Herbrand model*** of S.

• Let S be a set of clauses and class terms and H´ an untyped Herbrand interpretation of it. If every clause and definition in S is true according to H´ then H´ is an ***untyped Herbrand model*** of S.

2.1.5.3 Herbrand Interpretations and Semantical Structures

Given a semantical structure for a L-KARL language L with the alphabet (E, Σ, Boolean, Integer, Real, String, $A_{E\rightarrow}$, $A_{\Sigma\rightarrow}$, $A_{E\rightarrow\rightarrow}$, $A_{\Sigma\rightarrow\rightarrow}$, $\prod$, $A_{\prod}$, X_E, X_Σ, X_V), the corresponding Herbrand interpretation is the set of ground F-terms, P-terms, is-a terms, and definitions that are true in the structure. Therefore, if a set of clauses and definitions of L has a model it has a Herbrand model.

Conversely, for a Herbrand interpretation H the corresponding semantical structure,

$I = \langle U_E, U_\Sigma, U_V, \leq_U, I_U, I_{E\rightarrow}, I_{\Sigma\rightarrow}, I_{E\rightarrow\rightarrow}, I_{\Sigma\rightarrow\rightarrow}, I_{E\Rightarrow}, I_{\Sigma\Rightarrow}, I_{E\Rightarrow\Rightarrow}, I_{\Sigma\Rightarrow\Rightarrow}, I_{\prod}, I_{A\prod}\rangle$

is defined as follows (cf. [KLW90]):

- The domain U_E is $E^* \mid \cong$, i.e. the quotient[1] of E* induced by the equality terms in H. The equivalence class of a term T is denoted by $|T|$. $\|T\|$ denotes the representative of the equivalence class of T. Therefore $U_E := \{\|T\| \mid T \in E^*\}$.
- The domain U_Σ is Σ. The $T \in U_\Sigma$ are also denoted by $\|T\|$.
- The domain $U_V = U_{V,B} \cup U_{V,I} \cup U_{V,R} \cup U_{V,S}$ is the union of the four sets:
  - U_{VB} = {False, True};
  - U_{VI} is the set of all integers;
  - U_{VR} is the set of all real numbers;
  - U_{VS} is the set of all strings of finite length.

  $T \in U_V$ is also denoted by $\|T\|$.
- The ordering "$\leq_U$" of U is defined with the help of the is-a and is-element-of terms in H:
  - For all $e \in U_E$, $c \in U_\Sigma$, it holds $e \leq_U c$ if and only if $e\ \varepsilon\ c \in H$.
  - For all $c,c' \in U_\Sigma$, it holds $c \leq_U c'$ if and only if $c \leq c' \in H$.
  - $x \in U_{VB}$ is equivalent to $x \leq_U U_{VB}$.
  - $x \in U_{VI}$ is equivalent to $x \leq_U U_{VI}$.
  - $x \in U_{VR}$ is equivalent to $x \leq_U U_{VR}$.
  - $x \in U_{VS}$ is equivalent to $x \leq_U U_{VS}$.

1. As mentioned above, the equality terms define a congruence relation on all ground ID terms.

- $$I_U(u) = \begin{cases} \|u\| & \text{for every 0-ary function symbol } u \in E \\ u & \text{for every element } u \in \Sigma \\ u & \text{for every element } u \in V. \end{cases}$$
- $I_U(f)(\|T_1\|,..., \|T_k\|) = \|f(T_1,..., T_k)\|$, for every k-ary ($k > 0$) function symbol $f \in E$.
- $$I_{E\rightarrow}(a)(\|T\|) = \begin{cases} \|S\|, & \text{if } T[a : S] \in H,\ a \in A_{E\rightarrow} \\ \text{undefined} & \text{otherwise.} \end{cases}$$
- $$I_{\Sigma\rightarrow}(a)(\|T\|) = \begin{cases} \|S\|, & \text{if } T[a : S] \in H,\ a \in A_{\Sigma\rightarrow} \\ \text{undefined} & \text{otherwise.} \end{cases}$$
- $$I_{E\rightarrow\rightarrow}(s)(\|T\|) = \begin{cases} \{\|S\| \mid \text{where } T[s :: \{S\}] \in H\}, & \text{if } T[s :: \{\}] \in H,\ s \in A_{E\rightarrow\rightarrow} \\ \text{undefined} & \text{otherwise.} \end{cases}$$
- $$I_{\Sigma\rightarrow\rightarrow}(s)(\|T\|) = \begin{cases} \{\|S\| \mid \text{where } T[s :: \{S\}] \in H\}, & \text{if } T[s :: \{\}] \in H,\ s \in A_{\Sigma\rightarrow\rightarrow} \\ \text{undefined} & \text{otherwise.} \end{cases}$$
- $$I_{E\Rightarrow}(a)(\|T\|) = \begin{cases} \{\|S\| \mid \text{where } T[[\text{ELEMENT\_ATT } a : \{S\};]] \in H\}, & \text{if } T[[\text{ELEMENT\_ATT } a : \{\};]] \in H \\ \{\|S\| \mid \text{where } T'[[\text{ELEMENT\_ATT } a : \{S\};]] \in H\}, & \text{if } T'[[\text{ELEMENT\_ATT } a : \{\};]] \in H \text{ and } T \,\varepsilon\, T' \in H \\ \text{undefined,} & \text{otherwise.} \end{cases}$$
- $$I_{\Sigma\Rightarrow}(a)(\|T\|) = \begin{cases} \{\|S\| \mid \text{where } T[[\text{CLASS\_ATT } a : \{S\};]] \in H\} & \text{if } T[[\text{CLASS\_ATT } a : \{\};]] \in H \\ \text{undefined} & \text{otherwise.} \end{cases}$$
- $$I_{E\Rightarrow\Rightarrow}(a)(\|T\|) = \begin{cases} \{\|S\| \mid \text{where } T[[\text{ELEMENT\_ATT } s :: \{S\};]] \in H\}, & \text{if } T[[\text{ELEMENT\_ATT } s :: \{\};]] \in H \\ \{\|S\| \mid \text{where } T'[[\text{ELEMENT\_ATT } s :: \{S\};]] \in H\}, & \text{if } T'[[\text{ELEMENT\_ATT } s :: \{\};]] \in H \text{ and } T \,\varepsilon\, T' \in H \\ \text{undefined,} & \text{otherwise.} \end{cases}$$
- $$I_{\Sigma\Rightarrow\Rightarrow}(s)(\|T\|) = \begin{cases} \{\|S\| \mid \text{where } T[[\text{CLASS\_ATT } s :: \{S\};]] \in H\} & \text{if } T[[\text{CLASS\_ATT } s :: \{\};]] \in H \\ \text{undefined} & \text{otherwise.} \end{cases}$$
- $I_{\Pi}(p) = \{(\|T_1\|,..., \|T_k\|) \mid p(p_1 : T_1[],..., p_k : T_k[]) \in H\}$
- $I_{A\Pi}(p) = (T_1,..., T_k)$ with
  $T_i = \{\|T'_i\| \mid \text{PREDICATE } p\ p_1 : \{T'_1\};...; p_i : \{T'_i\};...; p_k : \{T'_k\}; \text{END} \in H\}$

These two results can be summed up by the following lemma.

Lemma 1

Let S be a set of clauses and definitions. S is unsatisfyable, i.e. it does not have a model, if and only if S has no Herbrand model.[1]

Remark 10 Untyped Herbrand interpretations and untyped semantical structures

Given an untyped semantical structure for a set of clauses and definitions, the corresponding untyped Herbrand interpretation is the set of ground F-terms, is-a terms, class terms, P-terms, and definitions that are true in the structure. Therefore, if the set of clauses and definitions has an untyped model it has an untyped Herbrand model. Conversely, for an untyped Herbrand interpretation H´, the corresponding semantical structure, $I = \langle U_E, U_\Sigma, U_V, \leq_U, I_U, I_{E\rightarrow}, I_{\Sigma\rightarrow}, I_{E\rightarrow\rightarrow}, I_{\Sigma\rightarrow\rightarrow}, I_{E\Rightarrow}, I_{\Sigma\Rightarrow}, I_{E\Rightarrow\Rightarrow}, I_{\Sigma\Rightarrow\Rightarrow}, I_\Pi, I_{A\Pi}\rangle$, can be constructed in the same way as was discussed above. Therefore, if S is a set of clauses and definitions, S has no untyped model if and only if S has no untyped Herbrand model.

2.1.6 Minimal Herband Model Semantics of L-KARL

In the last section the attention was restricted in two directions:

- First, only ***clauses*** are allowed instead of arbitrary formulae.
- Second, the set of all possible interpretations of a set of formulae was restricted to ***Herbrand interpretations*** and ***Herbrand models***.

These two restrictions correspond, because the following holds true for every set of clauses and definitions S:

S has a model $\Leftrightarrow$ S has a Herbrand model[2]

But currently no answer has been given to the simple question:

What is the meaning, i.e. the semantics, of a set of formulae and definitions S?

Two possible answers are possible:

- First, ***all*** Herbrand models represent the possible meanings of S.
- Second, a ***particular and definite*** Herbrand model is the meaning, i.e. the semantics, of S.

Because in general, there are infinite many Herbrand models the second possibility is favoured. In the following, the minimal Herbrand model is defined as semantics for a set of ***Horn clauses*** and definitions S. In the case of Horn clauses a ***definite least Herbrand model*** exists and will be considered as the intuitive semantics of a set of Horn clauses. Before this semantics is been introduced for L-KARL, this kind of semantics is briefly described for

1. The lemma is given in [KLW90].
2. cf. Lemma 1, at page 87.

predicate logic (cf. [Llo87]).

Def. 45 Horn clause

A ***definite Horn clause*** is a formula

$$\Lambda = \forall x_1 \ldots \forall x_s (l_1 \vee \ldots \vee l_m).$$

where each $l_1,\ldots, l_m$ is a literal, $x_1,\ldots, x_s$ are all variables occurring in $l_1 \vee \ldots \vee l_m$, and exactly one l_i is a positive literal.[1]

The Horn clause Λ can be written as:

$$\forall x_1 \ldots \forall x_s (l_i \leftarrow l'_1 \wedge \ldots \wedge l'_{i-1} \wedge l'_{i+1} \wedge \ldots \wedge l'_m).$$

where l_i is the original positive literal and all $l'_j = \neg l_j$, j=1,..., i-1, i+1,...,n, i.e. they are also positive literals. l_i is called the ***conclusion*** and the $l_{j(j\neq i)}$ are called the ***premises*** of the Horn clause.

In the following, the term Horn clause is used for definite Horn clauses.

Let the following set of Horn clauses in predicate logic be given:

$S = \{p(a), r(b), \forall x\ (q(x) \leftarrow p(x))\}$

Four possible Herbrand models are:

- $H_1 = \{p(a), q(a), r(b)\}$
- $H_2 = \{p(a), q(a), r(a), r(b)\}$
- $H_3 = \{p(a), q(a), q(b), r(a), r(b)\}$
- $H_4 = \{p(a), p(b), q(a), q(b), r(a), r(b)\}$

The first Herbrand model H_1 is ***minimal***. Each of its subsets would no longer be a model of S. This minimal Herbrand model corresponds to two intentions:

- ***Closed-world assumptions (cwa)***[2]: "... if no proof of a positive ground literal exists, then the negation of that literal is assumed true." [Cla78], p. 55. More formally:
  - If $S \models \varphi$ then $S \mid\approx \varphi$.
  - If $S \not\models \varphi$ then $S \mid\approx \neg\varphi$.
- ***Procedural interpretation of Horn clauses***: Every Horn clause

  $$\forall x_1 \ldots \forall x_s (l_i \leftarrow l'_1 \wedge \ldots l'_{i-1} \wedge l'_{i+1} \wedge \ldots \wedge l'_m).$$

  is interpreted as a procedure which computes for all variable bindings (ground instances) of the premise literals $l'_1 \wedge \ldots \wedge l'_{i-1} \wedge l'_{i+1} \wedge \ldots \wedge l'_m$ the bindings (ground instances) of the head l_i of the Horn clause. Only the ground instances of the head clause which can be inferred by the conjunction of the premise literals are considered as true. This is the same type of minimal criterion as is discussed for Herbrand models.[3] A fixpoint operator can be defined (cf. [Llo87], pp. 35ff) which can be used to compute this "procedural"

1. A Horn clause is a clause with one positive literal.
2. The closed-world assumption is the ignorant modern answer to Aristoteles: What we do not know, we know that it is not!
3. One can doubt whether this is really the intended meaning of a Horn clause. On the other hand, if this becomes common it will become a kind of self-fulfilling prophecy.

interpretation of a set of Horn clauses. This computed fixpoint corresponds to the least Herbrand model, which will be defined next.

Def. 46 Minimal Herbrand model for first-order logic

For a set of clauses S a partial ordering "≤" on all Herbrand interpretations of S is defined. Let H_1 and H_2 be two Herbrand interpretations of S then

$$H_1 \leq H_2$$

if and only if every ground positive literal which is an element of H_1 is also an element of H_2.

A Herbrand model H of S is ***minimal*** if and only if for every Herbrand interpretation H´ ≠ H

H´ ≤ H implies that H´ is no model of S.

Def. 47 Least Herbrand model

If a set of first-order clauses S has an unique minimal Herbrand model this model is called the ***least Herbrand model***.

Lemma 2 The intersection H of a non-empty set of Herbrand models of Horn clauses S is a Herbrand model of S.

Proof:

cf. [NiM90] and the proof for lemma 4 ∴

Lemma 3

Every set of first-order Horn clauses has a least Herbrand model.

Proof:

cf. [NiM90] and the proof for lemma 4 ∴

Remark 11 Inconsistency of the closed-world assumption in the case of arbitrary sets of clauses.

If the restriction on Horn clauses is omitted the closed-world assumption can lead to inconsistencies. In [Cla78], p.67, the following example is given:

$$S = \{p(a) \vee p(b)\}$$

- $S \not\models p(a)$ therefore $S \mid\approx \neg p(a)$
- $S \not\models p(b)$ therefore $S \mid\approx \neg p(b)$
- $S \models p(a) \vee p(b)$ therefore $S \mid\approx p(a) \vee p(b)$

But $\{p(a) \vee p(b), \neg p(a), \neg p(b)\}$ is an inconsistent set of clauses because there exists no model for it.

Remark 12 Least Herbrand model semantics and the closed-world assumption.

In the case of Horn clauses, the unique minimal Herbrand model, the so-called least Herbrand model, corresponds to the closed-world assumptions. Let φ be an arbitrary ground literal:

- $S \models \varphi$ means that every model of S is a model of φ. Therefore, the least Herbrand model H of S has to be a model of φ. Therefore, if the least Herbrand model H is a model of φ all other (greater) models of S are a model of φ.
- $S \not\models \varphi$ means that a model of S exists which is not a model of φ. This model does not contain φ. Therefore, the least Herbrand model H does not contain φ because it is a subset of the discussed model. Therefore, the least Herbrand model H assumes $\neg\varphi$ because $\varphi \notin H$, i.e. $H \models\!\approx \neg\varphi$.

In the case of clauses, it is possible that several minimal Herbrand models exist and therefore no least Herbrand model. The above discussed example $S = \{p(a) \vee p(b)\}$,[1] has two minimal Herbrand models:

- $H_1 = \{p(a)\}$
- $H_2 = \{p(b)\}$

This corresponds to the fact that the closed-world assumption can lead to inconsistencies for sets of arbitrary clauses.

Remark 13 Problem of the closed world assumptions

On the one hand, the closed-world assumption is not recursively enumerable because "$\models$" is not decidable.[2] On the other hand, if a ***finite*** least Herbrand model exists for a set of Horn clauses "$\models$" becomes decidable.

In order to allow operational semantics (i.e., to make KARL run or walk), we restricted our attention from all models to Herbrand models and from all Herbrand models to the least one. Whereas the first restriction does not change the semantics of the logical language the second one leads to a drastic change in the semantics. If one of the ground instances of φ is not in the least Herbrand model of a set of clauses S, i.e. if the least Herbrand model of S is not a model of φ, then

$$S \models\!\approx \neg\varphi$$

can be entailed.

Normally, in first-order logic one can only entail that

$$S \models \neg\varphi$$

if no model, i.e. no Herbrand model of S, is a model of φ, because every model of S is then a model of $\neg\varphi$.

1. $p(a) \vee p(b)$ is equivalent to $p(a) \leftarrow \neg p(b)$ or $p(b) \leftarrow \neg p(a)$.
2. A relation is recursively enumerable if it is the domain of a decidable relation (cf. [End72]).

Therefore, two main changes are carried out:

- The representation formalism is restricted to Horn clauses.
- The set of negative literals and therefore the set of formulae which we consider as logically entailed by a set of Horn clauses and definitions S is extended.

2.1.6.1 Extensions of Ordinary Minimal Herbrand Model semantics

The main problems in applying minimal Herbrand model semantics for L-KARL are caused by two reasons:

- L-KARL includes ***functional (single-valued) attributes***, i.e. two different values “u” and “u´” of an attribute for the same object have to be equal. Otherwise, only element-ID terms (E-ID terms) are allowed to be equal. Therefore, $u \cong u'$ is only possible, if “u” and “u´” are element denotations.
- L-KARL includes ***well-typing***, i.e. domain restrictions and range restrictions for attributes by class definitions and domain restrictions for predicates by predicate definitions. For example, the set of Horn clauses S = {o[a : t[]]} does not have a Herbrand model, because the well-typing condition is violated, i.e. their is no class term which allows the application of the attribute “a” on an element “o”.

In order to detect errors caused by the functionality of single-valued attributes or by typing, the following alternatives for a set of Horn clauses and definitions S are considered:

- It is checked to see whether S has an untyped Herbrand model.
  - If not, a error concerning equality terms (possibly caused by the functionality of single-valued attributes) is assumed.
  - Otherwise, the minimal Herbrand model of all untyped Herbrand model of S is considered as semantics.
- It is checked to see whether the minimal untyped Herbrand model is also a model of S, i.e. whether it is well-typed.
  - If not, a typing error is assumed.
  - Otherwise, it is considered as the semantics of S.

Before doing this formally, the definition of a Horn clause is slightly modified to prevent the derivation of new is-a terms.[1]

Def. 48 Horn clause

As a modification of Def. 45 “Horn clause” on page 88, the allowed conclusion of a Horn clause is restricted. The conclusion of a Horn clause has to be

- an F-term or

1. Cf. section 2.1.10.

• a P-term.

Def. 49 Minimal Untyped Herbrand model for L-KARL

For a set of clauses and definitions S a partial ordering on all untyped Herbrand interpretations of S is defined. Let $H_1´$ and $H_2´$ be two untyped Herbrand interpretations of S then

$H_1´ \leq H_2´$

if and only if every ground positive literal which is an element of $H_1´$ is also an element of $H_2´$.

An untyped Herbrand model H´ of S is ***minimal*** if and only if for every untyped Herbrand interpretation $H´´ \neq H´$

$H´´ \leq H´$ implies that H´´ is not an untyped Herbrand model of S.

Def. 50 Least untyped Herbrand model

If a set of clauses and definitions S has an unique minimal untyped Herbrand model this model is called the ***least untyped Herbrand model***. If S has no untyped Herbrand model, an error in respect to the equality of ID terms is assumed.

Lemma 4

Each set of Horn clauses and definitions S has a least untyped Herbrand model if it has an untyped model.

Proof:

As the proof is similar to the corresponding proof in predicate logic, only the significant parts are given.
We assume that S has an untyped model and therefore an untyped Herbrand model.
1. We show that the intersection H of two Herbrand models H´ and H´´ of S is also a Herbrand model of S.
Assuming that

$H = H´ \cap H´´$

is **not** a Herbrand model of S, then a Horn clause or a definition of S is therefore not true in H.

• S should contain a Horn clause with a ground instance

$A \leftarrow B_1 \wedge ... \wedge B_n$

which is not true in H. Therefore $B_i \in H$ for i = 1,..., n but $A \notin H$.
But these implies that

• $B_i \in H´$ for i = 1,..., n because $H \subseteq H´$ and

- $B_i \in H''$ for i = 1,..., n because $H \subseteq H''$ and
therefore $A \in H'$ and $A \in H''$ and therefore $A \in H$ and the Horn clause would be true in H.

- S should contain a definition which is not true in H. Because definitions "d" are ground, the following given equivalence holds:
$d \in H \Leftrightarrow H \models d$
Therefore, we would assume that
  - $d \notin H$
  - $d \in H'$
  - $d \in H''$

  which contradict to $H = H' \cap H''$.

2. We show that the intersection of all Herbrand models of S is a minimal Herbrand model.

- The intersection of all Herbrand models is again a Herbrand model of S, cf. (1).

- The intersection H of all Herbrand models is minimal, because otherwise a Herbrand model H' would exist with $H'' := H \cap H'$ and $H'' \neq H$ is a Herbrand model of S. On the other hand, H is the intersection of all Herbrand models. Therefore, this is not possible.

$\therefore$

The following definition sums up the minimal model semantics of a set of Horn clauses and definitions. It integrates well-typing and functionality of single-valued attributes (via equality errors) into a formal semantics. Instead of deriving the least Herbrand model H of a set of Horn clauses and definitions S as semantics two additional properties are we stipulated:

- Only if the least untyped Herbrand model H exists, i.e. it does not violate the functionality of single-valued attributes, is it assumed as semantics. Otherwise, an error in respect to equality is assumed.
- Only if the least untyped Herbrand model H is also a typed Herbrand model is it assumed as semantics. Otherwise, an error in respect to the well-typing condition is assumed. Therefore, no derivation of additional definitions is allowed which are not justified by class terms, is-a terms, and predicate definitions in S.

Def. 51 Well-typed least Herbrand model

- If the least untyped Herbrand model of a set of definitions and Horn clauses S does not exist, then an equality error is indicated
- If the least untyped Herbrand model of a set of definitions and Horn clauses S exists but is not a model of S, then a typing error is indicated.

- If the least untyped Herbrand model of a set of definitions and Horn clauses S exists and is also a model of S, then it is assumed to be the semantics of S and is called the ***well-typed least Herbrand model*** of S.

Instead of "|=" which is defined for all models, now the "non-monotonic" entailment "|≈" is defined which allows the entailment of formulae which concern only the well-typed least Herbrand model.

Def. 52 Entailment

Let S be a set of definitions and Horn clauses with a well-typed least Herbrand model H. Then S entails a clause Λ, written by

$$S \models_{\approx} \Lambda$$

if Λ is true in H, i.e. if H is a model of Λ.

2.1.7 Perfect Model Semantics of L-KARL

In the last section, the attention was limited to Horn clauses and the least Herbrand model was considered as semantics for a set of Horn clauses. In the following, this restriction is relaxed and again sets of clauses are regarded. This is done to allow the expression of negative information and the derivation of information based on negative information. For example, if one wants to express the fact that every human who is not a man is a woman, one can use the following clause:

$\forall x\ (\text{woman}(x) \leftarrow \text{human}(x) \wedge \neg\text{man}(x))$.

This clause is not a Horn clause, because in a Horn clause the head literal can only be dependent on positive literals.

As already discussed, arbitrary sets of clauses do not have a least Herbrand model but can have several minimal ones. If a minimal Herbrand model should used as semantics it must be defined which one of the minimal Herbrand models should be preferred. For this reason attention is limited to set of clauses which are ***stratified*** or ***locally stratified*** (cf. [ABW88], [Prz88], [Ull88]).

The rationale for applying ***stratification*** in predicate logic is to define a partial ordering of all predicates in order to define a selection criterion for the perfect model among all minimal models. Different strata, i.e. levels of the ordering relation, are defined and every predicate is assigned to one strata. Not all set of rules (i.e., clauses) can be ordered by stratification. For two predicate symbols "p" and "q" of a set of rules, the following relationship must hold:

$p \leq q$ if "p" is used as a premise of a rule containing "q" as a conclusion.

$p < q$ if "p" is used as a negated premise of a rule containing "q" as a conclusion.

The sets

$S = \{p(X) \leftarrow \neg q(X),\ q(X) \leftarrow \neg p(X)\}$ or

$S' = \{p(X) \leftarrow \neg p(X)\}$

cannot be ordered by stratification, for example. In general, a set of rules is stratified if the set P of its predicate symbols can be decomposed into disjoint sets $P_0, P_1, \ldots, P_r$, called strata, so that for every rule

$p \leftarrow \neg q_1 \wedge ... \wedge \neg q_m \wedge q_{m+1} \wedge ... \wedge q_{m+n}$
$p, q_1,..., q_{m+n}$ are positive literals
it holds true that

- the predicate symbol of the conclusion belongs to a strata “i”;
- the predicate symbols of the positive literals belong to the union of all strata which are less or equal than “i”;
- the predicate symbols of the negative literals belong to the union of all strata which are less than “i” (cf. [Prz88]).

The ***perfect model*** M is the minimal model with the following property: If M´ is any other minimal model, then for every ground fact of a predicate symbol $p \in M \setminus M´$ there is a ground fact of a predicate symbol $p´ \in M´ \setminus M$, such that $p´ < p$.

> “In fact, M is >>least << of all minimal model if one puts the most weight on having small set of ground facts of predicates at the lowest strata.” [Ull88], p. 139

Whereas stratification is defined for the predicate symbols used in a rule set, ***local stratification*** is defined for the Herbrand base of the rule set (cf. Def. 56 “Local stratification” on page 97). Local stratification is applied for L-KARL instead of normal stratification because every stratified rule set is locally stratified but not vice versa, i.e. it is a more general concept.

In the following, the local stratification and perfect Herbrand models are defined for L-KARL. The ***unique untyped perfect Herbrand model*** is introduced as its semantics. If this model exists, i.e. if a set of rules and definitions S has a model, and if this model does not violate the well-typing conditions, then it is a model of S. Otherwise, S has no model and an error is assumed. In the following this is introduced in a formal manner for L-KARL using F-logic (cf. [KLW90]]) as a model.

First a syntactical pattern is defined which must be used to write clauses. This will be used as a guide for selecting one minimal model of all possible ones. This restricted form of writing a clause is called a ***rule***.

Def. 53 Rule

A ***rule*** is a clause written as:

$\forall x_1 ... \forall x_s (l \leftarrow l_1 \wedge ... \wedge l_m)$.

In addition, l is not an is-a term, i.e. it is a F-term or P-term. Instances and ground instances of rules are defined exactly as they are defined for clauses.

The next four definitions (***atom, dependency graph, local stratification, stratification function***) are used to define a preference order of all minimal Herbrand models as it is done by [KLW90] for F-logic. For more details compare [ABW88], [Prz88], [Ull88].

Def. 54 Atom

- An equality term is its own ***equality atom***.

- An is-element-of term is its own ***is-element-of atom***.
- A flat data term
  $o[a_1 : T_1, ..., a_r : T_r, s_1 :: \{S_{11}, ..., S_{1m_1}\}, ..., s_t :: \{S_{t1}, ..., S_{tm_t}\}]$
  corresponds to the following set of ***data-atoms***:
  - $o[a_1 : T_1]$
  - ...
  - $o[a_r : T_r]$
  - $o[s_1 :: \{\}]$[1]
  - $o[s_1 :: \{S_{11}\}]$
  - ...
  - $o[s_1 :: \{S_{1m_1}\}]$
  - ...
  - $o[s_t :: \{\}]$
  - $o[s_t :: \{S_{t1}\}]$
  - ...
  - $o[s_t :: \{S_{tm_t}\}]$
- A ***F-atom*** is either an equality atom, is-element-of atom, or a data atom.
- Every flat P-term is its own ***P-atom***.

Def. 55 Dependency graph[2]

The ***dependency graph*** $DG_L(S')$ of a L-KARL language L and a set of rules S´ is defined as follows:

- Let S´* denote the set of all ground instances of S´ over L (cf. Def. 42 “Instance and ground instance” on page 84).
- The ***vertices*** of $DG_L(S')$ denoted by $N_L(S')$ correspond to the ground atoms constructable in L.
- A ***positive edge*** in $DG_L(S')$
  $\psi <=+= \phi_i$
  connects a pair of vertices $\psi, \phi_i \in N_L(S')$ if and only if there is a rule
  $\underline{\psi} \leftarrow \underline{\phi}_1 \wedge ... \wedge \underline{\phi}_i \wedge ... \wedge \underline{\phi}_k. \in S^*$
  such that ψ, ϕ_i are ground atoms of the positive literals $\underline{\psi}$ and $\underline{\phi}_i$, respectively.
- A ***negative edge*** in $DG_L(S')$
  $\psi <=-= \phi_i$
  connects a pair of vertices $\psi, \phi_i \in N_L(S')$ if and only if there is a rule

1. o[s :: { }] is not a tautology because of the well-typing condition.
2. cf. [KLW90]

$\underline{\psi} \leftarrow \underline{\phi}_1 \wedge \ldots \wedge \neg\underline{\phi}_i \wedge \ldots \wedge \underline{\phi}_k. \in S^*$

such that ψ, ϕ_i are ground atoms of the positive literals $\underline{\psi}$ and $\underline{\phi}_i$, respectively.

• In addition, $DG_L(S')$ has the following edges: whenever data atoms with functional attributes

$u[a : T_1]$ and $u[a : T_2]$,

where u, T_1, T_2 are ground ID terms with $T_1 \neq T_2$ and $a \in A_{E\rightarrow}$ (resp., $A_{\Sigma\rightarrow}$), are atoms of literals appearing in the heads of rules in S'^*, then

$T_1 \cong T_2$ <=+= $u[a : T_1]$

$T_1 \cong T_2$ <=+= $u[a : T_2]$

are positive edges in $DG_L(S')$.

• A directed path or a cycle in $DG_L(S')$ is ***negative*** if it contains at least one negative edge.

Def. 56 Local stratification

A set of rules S´ is ***locally stratified*** if:

• $DG_L(S')$ has no negative paths leading into equality atoms; and

• $DG_L(S')$ has no negative cycles.

Def. 57 Stratification function

If a set of rules S´ is locally stratified, a stratification function can be defined

level: $S'^* \rightarrow N$

such that if $DG_L(S')$ has a directed path from ψ to ϕ then

$\text{level}(\psi) \leq \text{level}(\phi)$;

if there is a negative path, then

$\text{level}(\psi) < \text{level}(\phi)$.

The function level can easy be extended to class terms, is-a terms, class definitions and predicate definitions by mapping them at the lowest value.

In the following the stratification is used to define an ordering on all minimal models of a set of rules and definitions S. In order to define the special model which is assumed to be the intended one out of the possibly infinite set of models a ***preference*** “<<“ on all models is introduced.

Def. 58 Perfect model

An untyped Herbrand Interpretation H_1' is preferable to a second one H_2', denoted by

$H_1' << H_2'$,

if whenever there is a ground atom ψ such that

$H_1' \models \psi$ and $H_2' \not\models \psi$

then there is a ground atom ϕ such that level(ϕ) < level(ψ), $H_2' \models \phi$, and $H_1' \not\models \phi$.

Models that are minimal with respect to "<<" are called ***perfect***.

The following lemma shows that every perfect model is a minimal model, i.e. the definition of a perfect model is a generalization of the minimal model semantics for sets of Horn clauses.

Lemma 5

Every untyped perfect model of a set of rules and definitions S is a minimal one.

Proof:

Let M denote a perfect model.
Assume that there exists a M´ which is smaller than the perfect model M, i.e. M´ is a proper subset of M.
At least one ground atom ψ exists with $\psi \in M$ and $\psi \notin M'$. Therefore $M \models \psi$ and $M' \not\models \psi$.
Because M is a perfect model, a ground atom ϕ with
level(ϕ) < level(ψ)
and $M \not\models \phi$ and $M' \models \phi$
must exist.
Therefore $\phi \in M'$ and $\phi \notin M$.
Therefore, M' is not a subset of M.
$\therefore$

Lemma 6

Every locally stratified set of rules and definitions S which has a model has a ***unique untyped perfect Herbrand model***.

Proof:

1. Every locally stratified set of L-KARL clauses can be mapped on a set of locally stratified clauses in predicate logic (cf. [KLW90]).
2. Every locally stratified set of clauses of predicate logic has an unique perfect model (cf. [Prz88]).
$\therefore$

Def. 59 Well-typed perfect Herbrand model

Let a set of definitions and rules S be given which is locally stratified.

- If the untyped perfect Herbrand model of S does not exist, an equality error is indicated
- If the untyped perfect Herbrand model of S exists but is not a model of S a typing error is indicated.

• If the untyped perfect Herbrand model of S exists and is also a model of S then it is assumed as the semantics of S and is called the ***well-typed perfect Herbrand model*** of S.

Def. 60 Entailment

Let S be a set of definitions and rules with a well-typed perfect Herbrand model H. Then S entails a clause Λ, written by

$$S \mid\approx \Lambda$$

if Λ is true in H, i.e. if H is a model of Λ (cf. Def. 52 “Entailment” on page 94).

Remark 14

The concept of stratified negation helps to select one model of the set of all minimal Herbrand models for a set of clauses. The previously discussed clause set $S = \{p(a) \vee q(b)\}$ has two minimal Herbrand models $H_1 = \{p(a)\}$ and $H_2 = \{q(b)\}$. Two possibilities for writing the disjunction as a rule are:

$R_1 = p(a) \leftarrow \neg q(b).$ or

$R_2 = q(b) \leftarrow \neg p(a).$

Both rules are logically equivalent to the disjunction

$p(a) \vee q(b).$

but both rules have different perfect Herbrand models. R_1 has H_1 and R_2 has H_2 as its perfect Herbrand model according to their stratification. The ***different syntactical structures of logically equivalent clauses*** results in different perfect Herbrand models as their semantics.

2.1.8 Constraints

As already discussed, a set of definitions and rules S must fulfil several constraints in order to have a perfect model (cf. Def. 59 “Well-typed perfect Herbrand model” on page 98):

- S must be ***locally stratified*** (cf. Def. 56 “Local stratification” on page 97).
- The untyped perfect Herbrand model must be free of ***equality errors*** (cf. Def. 38 “Equality error” on page 81).
- The untyped perfect Herbrand model must be well-typed, i.e. it need not contain a ***typing error*** (cf. Def. 19 “The Well-Typing Conditions” on page 70).

These conditions are hard-wired, i.e. they are fixed by the way the semantics for such a set S is defined. Additional conditions can be modelled which must be fulfilled by a model of S. They are not used for inferencing, i.e. to derive new information from given information but to ensure consistency. They do not extend the model of S but check it for integrity. The following definition defines a ***set of constraints*** which will be used to model such restrictions for a well-typed perfect Herbrand model of a set of definitions and rules S.

Def. 61 Set of necessary constraints

A ***set of constraints*** to a L-KARL language is a pair of two sets (C_1, C_2).

- C_1 is a set of definitions and rules over the L-KARL language.
- C_2 is a set of formulae over the L-KARL language.

Such a set of constraints (C_1, C_2) can be used to define ***correctness*** for a well-typed perfect Herbrand model of a set of definitions and rules S.

Def. 62 Correct well-typed perfect Herbrand model

A well-typed perfect Herbrand model H of a set of definitions and rules S is called a ***correct well-typed perfect Herbrand model*** of S according to a given set of constraints (C_1, C_2) if the following conditions hold:

- $H \cup C_1$ has a well-typed perfect Herbrand model.
- The well-typed perfect Herbrand model of $H \cup C_1$ is also a model of C_2.

If S has a correct well-typed perfect Herbrand model according to (C_1, C_2) then S ***satisfies*** the constraints (C_1, C_2).

A short example should help to clarify these definitions. First, a set of definitions and rules S is introduced which describes a graph using vertices and edges:

```
CLASS vertex
END

CLASS edge
        ELEMENT_ATT
                start : {vertex};
                end : {vertex};
END

N1 ε vertex, N2 ε vertex, N3 ε vertex,
N1[start : {N1}, end : {N2}] ε edge,
N2[start : {N2}, end : {N3}] ε edge,
N1[start : {N3}, end : {N1}] ε edge,
```

If one wants to express the fact that the graph should be acyclic, this can be done by the following set of constraints (C_1, C_2). First, C_1 "derives" the transitive closure of a relation "connected" which contains all pairs of connected vertices:

```
PREDICATE connected
        source : {vertex};
```

goal : {vertex};
END

$\forall s_E \ \forall x_E \ \forall y_E$
(connected(source : x_E, goal : y_E) $\leftarrow$ s_E[start : x_E, end : y_E] ε edge).

$\forall e_E \ \forall x_E \ \forall y_E \ \forall z_E$
(connected(source : x_E, goal : y_E) $\leftarrow$
e_E[start : x_E, end : z_E] ε edge $\wedge$ connected(source : z_E, goal : y_E)).

The well-typed perfect Herbrand model of $H \cup C_1$ would contain the following ground P-terms:

connected(source : N1, goal : N1)
connected(source : N1, goal : N2)
connected(source : N1, goal : N3)
connected(source : N2, goal : N1)
connected(source : N2, goal : N2)
connected(source : N2, goal : N3)
connected(source : N3, goal : N1)
connected(source : N3, goal : N2)
connected(source : N3, goal : N3)

The set C_2 of formulae defines the integrity conditions for the extended model:

$\forall x_E \ \neg$ connected(source : x_E, goal : x_E).

Because this formula is violated by connected(source : N1, goal : N1), for example, the well-typed perfect Herbrand model of $H \cup C_1$ is not a model of C_2. Therefore, S would **not** have a correct well-typed perfect Herbrand model.

2.1.9 Built-in Predicates

Arbitrary built-in predicates, i.e. predicates with predefined interpretations, can be introduced in L-KARL in order to improve the support of the language for modelling. In the following two built-in predicates “+” and “<“ are defined which can be used as a pattern for defining further ones.

Def. 63 “+“

“+“ is a three-ary predicate symbol and a P-term
+(argument$_1$: o_1, argument$_2$: o_2, sum : o_3)
is satisfied by a semantical structure I and a term assignment ν, denoted $I \models_\nu T$, if and only if the following conditions hold:

- $\nu(o_1), \nu(o_2), \nu(o_3) \in U_{V,I} \cup U_{V,R}$, and
- $\nu(o_1) + \nu(o_2) = \nu(o_3)$

Def. 64 “<“

“<“ is a two-ary predicate symbol and a P-term

$<(\text{less} : o_1, \text{greater} : o_2)$

is satisfied by a semantical structure I and a term assignment ν, denoted $I \models_\nu T$, if and only if the following given conditions hold:

- $\nu(o_1), \nu(o_2) \in U_V$, and $\nu(o_1)$ is lexicographically less than $\nu(o_2)$, or
- $\nu(o_1), \nu(o_2) \in U_\Sigma$, and $\nu(o_1)$ is lexicographically less than $\nu(o_2)$, or
- $\nu(o_1), \nu(o_2) \in U_E$, and $\nu(o_1)$ is lexicographically less than $\nu(o_2)$.

An alternative syntax for $<(\text{less} : o_1, \text{greater} : o_2)$ is $o_1 < o_2$ and an alternative syntax for $+(\text{argument}_1 : o_1, \text{argument}_2 : o_2, \text{sum} : o_3)$ is $o_1 + o_2 = o_3$. Every alphabet of a L-KARL language contains the predicate symbols “+” and “<“, and the argument names “argument_1”, “argument_2”, “greater”, “less”, and “sum”.

2.1.10 A Comparison of L-KARL with F-logic and O-logic

L-KARL is a restricted version of F-logic (cf. [KiL89], [KLW90]). Because F-logic does not distinguish between elements and classes it allows non-ground class terms, the derivation of new class denotations through the use of functions, and the derivation of new signature terms. F-logic is a powerful formal language for investigating the integration of object-orientation, functional and set-valued attributes, well-typing, and attribute inheritance in a well-defined semantical framework. Therefore we could use the main ideas of the semantics of F-logic for L-KARL. On the other hand, we customized F-logic in order to improve the modelling process with L-KARL:[1]

- We want to distinguish between objects which are classes and objects which are elements.
- We want to distinguish between attributes describing classes and those describing elements. This distinction is similar to the distinction of ***own*** and ***member slots*** in frame-based languages (cf. [FiK85]).
- We want to distinguish between an ***is-a***, i.e. a ***subset*** relationship, and an ***is-element-of*** relationship (cf. [Bra83]).
- We want to distinguish between ***object denotations*** indicating real-world objects and ***values*** which are used to describe real-world objects.
- We want to ***restrict the derivation of new class denotations and signature terms*** to make the language more handsome. Variation of terminological knowledge should not become possible as a result of new case-specific data. The terminological knowledge as it is

1. Even the authors of F-logic formulate the necessity for this kind of language-engineering (cf. [KLW90], p. 52).

represented by definitions is supposed to be static. For the same reason L-KARL does not allow variables as attribute names as F-logic does. F-logic allows non-ground ID terms as denotation of an attribute.

- In F-logic, attributes are considered as ***methods*** and the value of an attribute for an object could depend on other objects. Because up to now no application of KARL required this flexibility we have not introduced it into L-KARL.
- Our restriction of F-logic easily allows the definition of ***graphical representations*** for most of the modelling constructs, cf. chapter 4 of this thesis.

We had to pay a high price for these modifications because the semantics of L-KARL is less elegant than the semantics of F-logic. On the other hand, there seems to be a conflict between the epistemological adequacy of a language to support an easy modelling at the conceptual level and an elegant semantics. The semantics of F-logic is already quite more complex than the semantics of predicate logic without extending its expressive power. F-logic includes several additional modelling concepts which are not a part of predicate logic in order to improve the modelling process.

O-logic (cf. [KiW89], [KiW93]) and L-KARL are both restrictions of the full (syntactical) expressive power of F-logic and therefore share a lot of common features. The main differences are:

- O-logic does not allow classes as attribute values.
- Classes cannot described by data-terms, i.e. by attributes.
- O-logic does not integrate signature terms and well-typing.
- O-logic does not distinguish between objects and values.

Therefore, L-KARL is a less restricted version of F-logic than O-logic.

2.2 P-KARL: The Procedural Language of KARL

Experiences in XCON (cf. [SBJ87], [BaS89]) have shown that great problems arise if the control flow is only implicitly specified, e.g. in production rules. In KADS, knowledge about the control flow is clearly separated from other kinds of knowledge by describing it at a separate layer of the model of expertise (cf. [WSB92]). So P-KARL is used to explicitly describe knowledge about the control flow of a problem-solving method. By applying ***Procedural-KARL (P-KARL)*** the knowledge engineer can specify the control flow of a problem-solving method with terms familiar from common procedural languages, like Pascal or Modula-2 (cf. [Wir88]), for example. On the other hand, one goal of KARL is the declarative description of a model of expertise. This was solved by applying ***Dynamic Logic*** (cf. [Har79], [Har84], [Koz90]) and its semantics for P-KARL. This elegant method was learned from $(ML)^2$ (cf. [HaB92]) which is also a formal knowledge specification language based on KADS,[1] but we have chosen a different way of using Dynamic logic. Whereas in $(ML)^2$, inference actions are modelled by predicates of the dynamic logic and are embedded in the dynamic logic language, in KARL inference actions define the functions of a dynamic logic language. Therefore, the domain and inference layer define an interpretation for a P-KARL program (cf. chapter 3). A second difference between KARL and $(ML)^2$ is the restriction to ***deterministic while programs*** whereas $(ML)^2$ allows arbitrary programs.

In the following, the significant ideas of other approaches used by P-KARL, syntax, informal semantics, and formal semantics of P-KARL are introduced.

2.2.1 Significant Ideas of Other Approaches Used for P-KARL

Dynamic Logic (cf. [Har79], [Har84], [Koz90]) is a well-known means for declaratively describing procedural programs, i.e. for defining a model-theoretical semantics for them. 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. The execution of a program changes the value assignments of some variables as a result.
- Formulae: A formula is true for some states and false for the other ones.

Simple primitive programs have the form:

$$y := f(x)$$

and change 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. Complex programs can be built up by other programs and the operators “$\cup$”, “;”, and “*”.

1. cf. chapter 6.

- "$p \cup q$", with "p" and "q" being programs, models non-deterministic choice, i.e. either "p" or "q" are executed. If p^A and q^A denote the relations which interpret the programs p and q then $(p \cup q)^A = p^A \cup q^A$, i.e. the semantics of this composed program is the union of the two relations which interpret their single components.
- "p;q", with "p" and "q" being programs, models sequence, i.e. first "p" and then "q" are executed. If p^A and q^A denote the relations which interpret the programs p and q then $(p;q)^A = p^A \bullet q^A$, i.e. the semantics of this composite program is the composition of the two relations which interpret their single components "p" and "q".
- "p*", with "p" are program, models non-deterministic iterations. If p^A denotes the relation which interprets the programs represented by p then $(p^*)^A$ is the transitive closure of the relation p^A.

An additional component of dynamic logic consists of logical formulae. According to the expressive power of these formulae, different types of dynamic logic, i.e. propositional and first-order dynamic logic, can be distinguished. In the following, the case of first-order formulae without quantifiers is discussed. 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 $x^A \in P^A$, i.e. where the value assignment of the variable x is an element of relation which interprets the predicate symbol P. Complex formulae can be built up by logical connectives "$\neg$", "$\vee$", and "$\wedge$".

The last ingredient which is introduced is the test-operator "?" which connects formulae with programs. If "ψ" is a formula then "ψ?" is a program which is interpreted by the binary relation "$(\psi?)^A$" which contains all tuples for which the formula is satisfied:

$$(\psi?)^A = \psi^A \times \psi^A$$

The test-operator can be used to define conditions for the control flow of a program. A program "$\psi?;p \cup \neg\psi?;q$" has as semantics all tuples $(\nu_1, \nu_2) \in p^A$ with $\nu_1 \in \psi^A$ or $(\nu_1, \nu_2) \in q^A$ with $\nu_1 \notin \psi^A$. Therefore, it 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*** (cf. [Koz90]) because this restricted expressive power is sufficient to model the task layer of a KARL specification (cf. section 3.3).

- P-KARL allows only formulae without quantifiers, i.e. the alphabet of P-KARL does not contain quantifiers. Therefore, P-KARL programs are ***regular*** programs.
- The following condition restricts the allowed programs to a subset of all regular programs: P-KARL allows only deterministic while programs, i.e. a program has to be regular, and choice and iteration can only be used in a restricted manner (cf. Def. 67 "Program" at page 107).

In the following, syntax and semantics of P-KARL are defined in a formal manner.

2.2.2 Syntax of P-KARL

In the next definition, an ***alphabet of a P-KARL language*** is defined. It contains all elements which are used to form expressions of a P-KARL language. It consists of function symbols, variables, propositions, logical operators, program operators, and a test operator. Then, all ***formulae*** are defined which can be made up of predicate symbols, variables, and logical operators.

Def. 65 Alphabet of a P-KARL language

An ***alphabet*** P = (F, X, B, PS) of a P-KARL language consists of

- A finite set $F = \{f_1, f_2,..., f_r\}$ of k-ary (k > 0) ***function symbols***.
- The 0-ary ***function symbols*** FALSE and TRUE.
- A finite set $X = \{x_1,..., x_n\}$ of ***dataflow variables*** and a finite set $B = \{b_1,..., b_m\}$ of ***control flow variables***.
- A finite set of ***predicate symbols*** $PS = \{P_1,..., P_s\}$ with fixed arity.
- The ***logical operators*** "$\neg$", "$\vee$", and "$\wedge$".
- The ***program operators***
  - $\cup$ non-deterministic choice
  - ; sequence
  - * iteration.
- The ***test operator*** "?".

Def. 66 Formulae

- An ***atomic formula*** ψ has the form $P(x_1,..., x_j)$ with $P \in PS$, $x_i \in X \cup B$ for all i = 1,..., j, and j is the arity of P.
- ***Composed formulae*** are defined by mutual induction as follows. If ψ and ϕ are formulae, then
  - $\neg\psi$
  - $\psi \vee \phi$
  - $\psi \wedge \phi$

  are formulae.
- These are all formulae.

In the next definition all ***programs*** are defined which can be formed by assignments, program operators, and the test operator.

Def. 67 Program

- ***Primitive Programs*** are ***assignments*** of the form:
  - $(y_1, ..., y_{j_i}) := f_i(x_1, ..., x_{k_i})$
  with:
    - $y_{p_q} \in X$ for $p_q = 1,..., j_i$,
    - $y_{p_q} \neq y_{r_v}$ for $p_q, r_v \in \{1,..., j_i\}$ and $p_q \neq r_v$,
    - $x_{p_q} \in X$ for $p_q = 1,..., k_i$,
    - $x_{p_q} \neq x_{r_v}$ for $p_q, r_v \in \{1,..., k_i\}$ and $p_q \neq r_v$, and
    - $f_i \in F$;
  - $b := \mu$
  where $b \in B$ and μ = FALSE or μ = TRUE.
- ***Programs*** are defined by mutual induction, as follows: If "p", "q" are programs and ψ is a formula, then
  - $\psi?;p \cup \neg\psi?;q$,
  - $(\psi?;p)^*;\neg\psi?$, and
  - $p;(\neg\psi?;p)^*;\psi?$
  are programs.
- These are all programs.

Because the syntactical structure of P-KARL programs is defined in the restricted manner of the above definition it is ensured that P-KARL programs are deterministic (cf. section 2.2.5). All P-KARL programs can therefore be composed of if-then-else branches and while-do or repeat-until loops.

Remark 15 A second notation for elementary programs

Instead of

$$(y_1, ..., y_{j_i}) := f_i(x_1, ..., x_{k_i})$$

we also write

$$(y_1 := {}^1f_i(x_1, ..., x_{k_i}), ..., y_{j_i} := {}^{j_i}f_i(x_1, ..., x_{k_i})) .$$

Because the notion of a program written in dynamic logic is not very common we introduce some "syntactical sugar" which improves the readability of programs for people used to work with procedural programming languages.

Def. 68 Additional syntactical sugar for P-KARL

- For $\psi?;p \cup \neg\psi?;q$ it is written: **IF** ψ **THEN** p **ELSE** q **ENDIF**
- For $(\psi?;p)^*;\neg\psi?$ it is written: **WHILE** ψ **DO** p **ENDDO**
- For $p;(\neg\psi?;p)^*;\psi?$ it is written: **REPEAT** p **UNTIL** ψ

- A ***subroutine*** defines a name for a program “p”. The syntactical structure looks as follows:

  SUBTASK name
  p
  ENDSUBTASK

  If a program “q” uses “p” as a component, it can use the name of the subroutine instead of “p”. The semantics of a subroutine is defined by the semantics of “p”. The name of a subroutine defines just a macro, i.e. P-KARL does not have a block concept, local variables, or means like “call by reference” or “call by value”. All variables of a P-KARL program are global variables.

The last definition introduces the term initialized program which is used to determine a specific state as start for a program.

Def. 69 Initialized programs

Let “p” be a program, then
init; p
is an ***initialized program***. These are all initialized programs.

2.2.3 Informal semantics of P-KARL

Dynamic logic is a well investigated means for declaratively describing procedural languages. The key idea is to describe a program by means of a binary relation between its input and output. This relation describes the output for every given input. The state of a program is described by the values of all variables of a program. The “execution” of a program maps this state on a successor state according to the relation which describes the program. A primitive program is a value assignment. It changes the program´s state by assigning a new value to a variable. Programs can be constructed from primitive programs by use of the well-known control structures:

- If-then-else branch,
- while-do loop,
- repeat-until loop, and
- sequence.

The relation which interprets a program is a composition of the relations used to interpret the primitive programs it is composed of. The gist of the semantics of programs is the definition of the meaning of an elementary program like y := f(x). The binary relation $(y := f(x))^A$ which interprets this elementary program is the set of tuples (ν,μ) where ν is an arbitrary variable assignment and μ is a variable assignment with the properties:

- $\nu(z) = \mu(z)$ for all variables $z \neq y$, i.e. ν and μ define the same variable assignment for all variables which are not equal to y.

- $\mu(y) = f^A(\nu(x))$, i.e. μ differs from ν for the assignment of y. The assignment is defined by the function f^A which interprets the function symbol f applied to the value $\nu(x)$.

In chapter 3 it is shown how the dataflow variables correspond to the stores of an inference layer, i.e. how every variable corresponds to one store s_i. The value of this assignment is computed by a function "f" which will be defined by means of an elementary inference action. The new assignment of an output store is computed according to the given values of the variables. Subroutines correspond to composed inference actions. Formulae are provided to define conditions for branches and loops. In addition, the control flow variables are introduced which can be used to store truth values.

2.2.4 Semantics of P-KARL

First, a semantic structure and a variable assignment are defined which are used to interpret a P-KARL program. A variable assignment is also called a state, because it characterizes the state of a program by the values which are assigned to every variable.

Def. 70 Semantical structure of P-KARL

A ***semantical structure*** A for a P-KARL language P = (F, X, B, PS) is a tuple

$$A = (\wp(D), F^A = \{f_1{}^A,..., f_r{}^A\}, PS^A = \{P_1{}^A,..., P_s{}^A\})$$

- D is a non-empty set and $\wp(D)$ denotes the power set of D. $\wp(D)$ is called the ***carrier*** of A.
- All the $f_i{}^A \in F^A$ are used to interpret the function symbols. Every $f_i{}^A$, i = 1,...,r, is a function of the cartesian product of $\wp(D)$.

  $$f_i{}^A\text{: } \wp(D) \text{ x ... x } \wp(D) \rightarrow \wp(D) \text{ x ... x } \wp(D)$$

  The arity of the domain and range of the functions are determined by the arities of the corresponding function symbols $f_i \in F$. The 0-ary function FALSE is interpreted by the set $\{\emptyset\}$ and the 0-ary function symbol TRUE is interpreted by the set $\wp(D)$.
- All the $P_i{}^A \in PS^A$ are used to interpret the predicate symbols. Every $P_i{}^A$, i = 1,...,s, is a relation over the cartesian product of $\wp(D)$.

  $$P_i{}^A \subseteq \wp(D) \text{ x ... x } \wp(D)$$

  The arities of the relations are determined by the arities of the corresponding predicate symbols $P_i \in PS$.

A semantical structure for a P-KARL language interprets every function symbol of the alphabet by means of a function and every predicate symbol by means of a relation. The 0-ary function symbols are interpreted by 0-are functions. All other function symbols are interpreted as function over the cartesian product of the power set of a set D. Next, variable and term assignment are defined.

Def. 71 Variable assignments, term assignments, and states

- A ***variable-assignment***, ν, is a mapping

  $\nu: X \cup B \to \wp(D)$

  $\nu(x) = x^A$

- A variable-assignment is also called a ***state***. The set of all possible variable assignments of A is denoted by S^A.

- A variable-assignment is extended to a ***term assignment*** $f_i(x_1, ..., x_{k_i})$ in the following way:

  $$\nu(f_i(x_1, ..., x_{k_i})) = f_i^A(\nu(x_1), ..., \nu(x_{k_i}))$$

Now, the truth value of formulae and the meaning of programs can be defined by interpreting formulae as subsets of all states (i.e., these subsets define all states for which a formula is true) and programs as binary relations between states.

Def. 72 Satisfaction of a formula

With every formula ψ a subset $\psi^A \subseteq S^A$ is associated. Every variable assignment $\nu \in \psi^A$ satisfies the formula, i.e. a formulae ψ is ***satisfied*** by a semantic structure A and a term assignment ν, denoted

$A,\nu \models \psi$

if and only if

$\nu \in \psi^A$.

- An ***atomic formula*** $P(x_1,..., x_j)$ is satisfied by a semantic structure A and a term assignment ν ($A,\nu \models P(x_1,..., x_j)$) if and only if $(\nu(x_1),..., \nu(x_j)) \in P^A$.
- The sets ψ^A, ϕ^A for ***composed formulae*** are defined by mutual induction according to the structures of ψ and ϕ.
  - $(\neg\psi)^A = S^A - \psi^A$
  - $(\psi \vee \phi)^A = \psi^A \cup \phi^A$
  - $(\psi \wedge \phi)^A = \psi^A \cap \phi^A$.

In L-KARL, all formulae are closed. In P-KARL, all formulae are open. Therefore, the truth value of a L-KARL formula is independent of a variable assignment, whereas the truth value of a P-KARL formula can only be determined according to a given variable assignment.

Def. 73 Meaning of a program

With every program "p" a binary relation $p^A \subseteq S^A \times S^A$ is associated.
The sets p^A are defined by mutual induction according to the structure of "p" and ψ.

- $(y_1 := {}^1f_i(x_1, ..., x_{k_i}), ..., y_{j_i} := {}^{j_i}f_i(x_1, ..., x_{k_i}))^A = \{(\nu, \nu[f_i]) \mid \nu \in S^A\}$

$\nu[f_i]$ is a term assignment with:

$$\nu[f_i](x) = \begin{cases} {}^tf_i^A(\nu(x_1), ..., \nu(x_{k_i})) & \text{if } x = y_t \text{ for } t = 1,..., j_i \\ \nu(x) & \text{else.} \end{cases}$$

- $(b := \text{TRUE})^A = \{(\nu, \nu[\text{TRUE}]) \mid \nu \in S^A\}$ with

$\nu[\text{TRUE}]$ being a term assignment with:

$$\nu[\text{TRUE}](x) = \begin{cases} \nu(x) & \text{if } x \neq b \\ \wp(D) & \text{if } x = b \end{cases}$$

- $(b := \text{FALSE})^A = \{(\nu, \nu[\text{FALSE}]) \mid \nu \in S^A\}$ with

$\nu[\text{FALSE}]$ being a term assignment with:

$$\nu[\text{FALSE}](x) = \begin{cases} \nu(x) & \text{if } x \neq b \\ \{\emptyset\} & \text{if } x = b \end{cases}$$

- $(p;q)^A = \{(\mu, o) \mid \exists \nu \in S^A \text{ with } (\mu, \nu) \in p^A \text{ and } (\nu, o) \in q^A\}$.
- $(p \cup q)^A = p^A \cup q^A$.
- $(p^*)^A$ = the reflexive transitive closure of p^A

The reflexive transitive closure of a binary relation p^A is $p^A{}_0 \cup p^A{}_1 \cup p^A{}_2 \cup ...$
with

$p^A{}_0 = \{(\nu, \nu) \mid \nu \in S^A\}$ and
$p^A{}_{i>0} = \{(\nu, \mu) \mid \exists o \in S^A \text{ with } (\nu, o) \in p^A{}_{i-1} \text{ and } (o, \mu) \in p^A\}$.

- $(\psi?)^A = \{(\nu, \nu) \mid I, \nu \models \psi\}$.

As already mentioned, the semantics of an elementary program

$y := f(x)$ with $x, y \in X$

is the set of tuples (ν, μ) where ν is an arbitrary variable assignment and μ is a variable assignment with the properties:

- $\nu(z) = \mu(z)$ for all variables $z \neq y$, i.e. ν and μ define the same variable assignment for all variables which are not equal to y.
- $\mu(y) = f^A(\nu(x))$, i.e. μ differs from ν for the assignment for y. The assignment is defined by the function f^A which interprets the function symbol f applied to the value $\nu(x)$ which is assigned to x according to ν.

The semantics of an elementary program

b := TRUE (resp., b := FALSE) with $b \in B$

is the set of tuples (ν, μ) where ν is an arbitrary variable assignment and μ is a variable assignment with the properties:

- $\nu(z) = \mu(z)$ for all variables $z \neq b$, i.e. ν and μ define the same variable assignment for all variables which are not equal to b.
- $\mu(b) = \wp(D)$ (resp., $\mu(b) = \{\emptyset\}$), i.e. μ differs from ν for the assignment of b.

The semantics of composite programs is constructed according to the semantics of their components and the way these components are connected. In the case of a ***sequence***

p;q

the semantics of the composite program is the composite of the two relations which interprets its components, i.e.

$$(p;q)^A = p^A \bullet q^A$$

The semantics of a ***choice*** is the union of the interpretations of their components and the semantics of an ***iteration*** is the transitive closure of the relation which interprets its components. In the next subsection it is proved, that each choice and iteration in P-KARL is deterministic, i.e. the "execution" of such a program produces a definite result.

The last kind of composite programs, i.e. $\psi?$, define conditions for a program flow. Only the variable assignments which satisfy the formula ψ are considered in the further program flow.

The following definition is used to define an initial variable assignment for a P-KARL program. All variables are assigned to $\{\emptyset\}$.

Def. 74 Meaning of a initialized program

With every initialized program

init; p,

a binary relation

$$(init;\, p)^A \subseteq S^A \text{ x } S^A$$

is associated with

$$init^A = \{(\mu,\nu) \mid \mu \in S^A, \nu \in S^A \text{ with } \nu(x) = \{\emptyset\} \text{ for } x \in X \cup B\}$$

2.2.5 P-KARL Programs are Deterministic

In general, dynamic logic allows non-deterministic programs. The same input can lead to different possible output states of a program. Because P-KARL programs are supposed to be deterministic, we show in the following, how P-KARL restricted dynamic logic so, that if it stops every program stops in a state which depends functionally on the initial state. We cannot show that a program always stops. By doing this, the semantics of the syntactical constructs of Def. 68 "Additional syntactical sugar for P-KARL" at page 107 is illustrated.

Remark 16 Let "p" be a program and $\{(\nu,\mu) \mid (\nu,\mu) \in p^A\} \subseteq S^A \text{ x } S^A$ its interpretation. Then

μ depends functional on ν if it exists.[1]

This is proved by showing that P-KARL programs are ***deterministic***, i.e. the application of a program on a state can only result in an unique successor state or in no state. This is done by mutual induction according to the structure of “p”.

- If “p” is a primitive program $(y_1, ..., y_{j_i}) := f_i\,(x_1, ..., x_{k_i})$
  $(y_1 := {}^1f_i\,(x_1, ..., x_{k_i}), ..., y_{j_i} := {}^{j_i}f_i\,(x_1, ..., x_{k_i}))^A = \{(\nu,\mu) \mid (\nu,\mu) \in p^A$ with
  - $\mu(x) = {}^tf_i^A(\nu(x_1),..., \nu(x_{k_i}))$ for $x = y_t$ and $t = 1,..., j_i$ and
  - $\nu(x) = \mu(x)$ else$\}$

  therefore μ depends functionally on ν because the ${}^tf^A$ for $t = 1,..., j_i$ are functions.
- If “p” is a primitive program $b_j := \rho$
  $(b_j := \rho)^A = \{(\nu,\mu) \mid (\nu,\mu) \in p^A$ with $\mu(b_j) = \rho^A$ and $\nu(x) = \mu(x)$ else$\}$
  therefore μ depends functional on ν.
- **IF** ψ **THEN** q **ELSE** r **ENDIF**
  If $p = \psi?;q \cup \neg\psi?;r$ this follows because either “q” or “r” can be used to compute the successor state μ of a state ν. If “q” and “r” are functional, then every state μ with $(\nu,\mu) \in p^A$ must depend functionally on $\nu \in S^A$ because it depends either on “q” or on “r”. It is not possible that ψ and $\neg\psi$ are evaluated to be true according to the same state $\nu \in S^A$ (compare the semantics of “$\neg$“).
- **WHILE** ψ **DO** q **ENDDO**
  If $p = (\psi?;q)^*;\neg\psi?$ and it is assumed that a tuple $(\nu,\mu) \in ((\psi?;q)^*;\neg\psi?)^A$ exists, then it must hold true that certain states $\nu_0, \nu_1,..., \nu_n$, $0 \le n$, exist such that
  - (1) $\nu = \nu_0$,
  - (2) $\mu = \nu_n$,
  - (3) if $n > 0$: $\nu_i \in \psi^A$ for $i = 0,..., n-1$,
  - (4) if $n > 0$: $(\nu_i, \nu_{i+1}) \in q^A$ and because q is deterministic, ν_{i+1} depends functionally on ν_i and therefore ν_{i+1} depends functionally on $\nu_0 = \nu$, and
  - (5) $\nu_n \in (\neg\psi)^A$, i.e. $\nu_n \notin \psi^A$

  If μ exist, then μ is unique. This is proved by showing that n is unique.
  - If $\mu = \nu_0$, i.e. $n = 0$, then $(\nu_0,\nu_0) \in ((\psi?;q)^*;\neg\psi?)^A$. Therefore $(\nu_0,\nu_1) \notin ((\psi?;q)^*;\neg\psi?)^A$ because
    - $(\nu_0,\nu_0) \in ((\psi?;q)^*;\neg\psi?)^A$ and (5) requires that $\nu_0 \notin \psi^A$
    - $(\nu_0,\nu_1) \in ((\psi?;q)^*;\neg\psi?)^A$ and (3) requires that $\nu_0 \in \psi^A$.
  - If $\mu = \nu_j$, i.e. $n = j$, then $(\nu_0,\nu_j) \in ((\psi?;q)^*;\neg\psi?)^A$. Therefore $(\nu_0,\nu_{j+1}) \notin ((\psi?;q)^*;\neg\psi?)^A$ because
    - $(\nu_0,\nu_j) \in ((\psi?;q)^*;\neg\psi?)^A$ and (5) requires that $\nu_j \notin \psi^A$
    - $(\nu_0,\nu_{j+1}) \in ((\psi?;q)^*;\neg\psi?)^A$ and (3) requires that $\nu_j \in \psi^A$.

1. We do not know whether this element exists because we do not know whether loops always stop. The result of an initialization and a P-KARL program could be an infinite loop.

Therefore μ is unique and because of (2) and (4) (resp., because of (1) for $n = 0$) μ depends functional on ν.

- **REPEAT** q **UNTIL** ψ
  The same holds for $p = q;(\neg\psi?;q)^*;\psi?$.

2.3 KARL = P-KARL Interpreted by L-KARL

In the following, the global semantics of a KARL program is defined and a brief illustration is given. A further example can be found in chapter 3 and 5 of this thesis. In chapter 3, the general idea of this subsection is modified (i.e., customized) according to the epistemological primitives of KADS (cf. [WSB92]).

2.3.1 The Semantics of a KARL program

Let a P-KARL language P with an alphabet

$(F = \{f_1, f_2,..., f_r\}, X = \{x_1,..., x_n\}, B= \{b_1, b_2,..., b_m\}, PS)$

and a L-KARL language L with the alphabet

$(E, \Sigma, \text{Boolean, Integer, Real, String}, A_{E\rightarrow}, A_{\Sigma\rightarrow}, A_{E\rightarrow\rightarrow}, A_{\Sigma\rightarrow\rightarrow}, \prod, A_{\prod}, X_E, X_\Sigma, X_V)$

be given. ***L is used to define a semantical structure for P***.

In the following, it is discussed how a semantical structure

$A = (\wp(D), F^A = \{f_1{}^A,..., f_r{}^A\}, PS^A = \{P_1{}^A,..., P_s{}^A\})$

is defined by a L-KARL language L and several sets of rules and definitions using L.

2.3.1.1 The Carrier

D is defined by the Herbrand base HB of L.

$D := HB$

$\wp(D) := \wp(HB)$.

Therefore, a variable assignment ν maps every variable $x \in X \cup B$ on a subset $\nu(x)$ of the Herbrand base HB, i.e. the assignment of a variable is a set of ground literals and definitions.

2.3.1.2 The Dataflow Variables

With every dataflow variable $x_i \in X$ a set of L-KARL class definitions CD_i, i = 1,..., n are associated. They are used to restrict the possible value assignments of these variables. The following definition ***of terms over a set of class definitions*** is used to define subsets of the correct well-typed perfect Herbrand model of a set of L-KARL rules and definitions as a value assignment for a dataflow variable $x \in X$.

Def. 75 Terms over a set of class definitions

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

- An is-element-of term $e \,\varepsilon\, c \in H$ is a term over the class definitions CD, i.e.
  $e \,\varepsilon\, c \in CD(H)$,
  if and only if a class term c[[]] exists with CD |= c[[]].[1]
- A data term $e[a_1 : T_1,..., a_r : T_r, s_1 :: \{S_{11},..., S_{1m_1}\},..., s_t :: \{S_{t1},..., S_{tm_t}\}] \in H$, "e" is an E-ID term, is a term over the class definitions CD, i.e.
  $e[a_1 : T_1,..., a_r : T_r, s_1 :: \{S_{11},..., S_{1m_1}\},..., s_t :: \{S_{t1},..., S_{tm_t}\}] \in CD(H)$,
  if and only if a class term
  c[[ELEMENT_ATT $a_1 : \{\},..., a_r : \{\}, s_1 :: \{\},..., s_t :: \{\}$;]] exists with
  CD |= c[[ELEMENT_ATT $a_1 : \{\},..., a_r : \{\}, s_1 :: \{\},..., s_t :: \{\}$;]] and $e \,\varepsilon\, c \in H$.
- A data term $c[a_1 : T_1,..., a_r : T_r, s_1 :: \{S_{11},..., S_{1m_1}\},..., s_t :: \{S_{t1},..., S_{tm_t}\}] \in H$, "c" is a C-ID term, is a term over the class definitions CD, i.e.
  $c[a_1 : T_1,..., a_r : T_r, s_1 :: \{S_{11},..., S_{1m_1}\},..., s_t :: \{S_{t1},..., S_{tm_t}\}] \in CD(H)$,
  if and only if a class term
  c'[[CLASS_ATT $a_1 : \{\},..., a_r : \{\}, s_1 :: \{\},..., s_t :: \{\}$;]] exists with
  CD |= c'[[CLASS_ATT $a_1 : \{\},..., a_r : \{\}, s_1 :: \{\},..., s_t :: \{\}$;]] and CD |= $c \le c'$.
- An equality term $o \cong o' \in H$ is a term over the class definitions CD, i.e.
  $o \cong o' \in CD(H)$.

2.3.1.3 The Functions

$F^A = \{f_1{}^A, f_2{}^A,..., f_r{}^A\}$ is defined as a set of partial functions by

$$f_i{}^A\colon \wp(HB)^{k_i} \to \wp(HB)^{j_i}$$

and an elementary assignment

$$(y_1 := {}^1f_i(x_1, ..., x_{k_i}), ..., y_{j_i} := {}^{j_i}f_i(x_1, ..., x_{k_i}))^A$$

is interpreted by (cf. Def. 73 "Meaning of a program" at page 111)

$\{(\nu,\nu[f_i]) \mid \nu \in S^A\}$, $\nu[f_i]$ is a term assignment with:

$$\nu[f_i](x) = \begin{cases} {}^tf_i^A(\nu(x_1), ..., \nu(x_{k_i})) & \text{if } x = y_t,\ t = 1,..., j_i \\ \nu(x) & \text{else.} \end{cases}$$

The functions ${}^tf_i{}^A\colon \wp(HB)^{k_i} \to \wp(HB)$ remains to be defined. For this purpose, r sets S_i of rules and definitions of L must be defined. Their Herbrand models according to a given variable assignment ν are used to define such interpretation functions.

1. Cf. Def. 3 "Class term" at page 49.

Def. 76 Interpretation function

Let r sets S_i of rules and definitions over L be defined and a variable assignment $\nu \in S^A$ be given. An interpretation function is a partial function ${}^t f_i^A$:

$${}^t f_i^A: \wp(HB)^{k_i} \rightarrow \wp(HB)$$
$$\nu[f_i](y_t) = {}^t f_i^A(\nu(x_1),..., \nu(x_{k_i}))$$
$$\text{for } t = 1,..., j_i \text{ and } i = 1,..., r$$

${}^t f_i^A$ defines the set of all term of the correct well-typed perfect Herbrand model H:

$$H = \nu(x_1) \cup ... \cup \nu(x_{k_i}) \cup S_i$$

over the class definitions which are associated with y_t as the new assignment $\nu[f_i]$ for y_t. If this correct well-typed perfect Herbrand model H does not exist, then the interpretation function $f_i^A(\nu(x_1), ..., \nu(x_{k_i}))$ is not defined for $(\nu(x_1), ..., \nu(x_{k_i})) \in \wp(HB)^{k_i}$.

2.3.1.4 The Relations

For every predicate symbol $P \in PS$, a corresponding relation $P^A \subseteq \wp(HB) \times ... \times \wp(HB)$ is defined. Therefore, formulae of a P-KARL language can be used to formally reason about properties of perfect Herbrand models of sets of L-KARL clauses.

2.3.2 An Example

The simple version of heuristic classification as introduced in chapter one is used as an illustration (see figure 8).

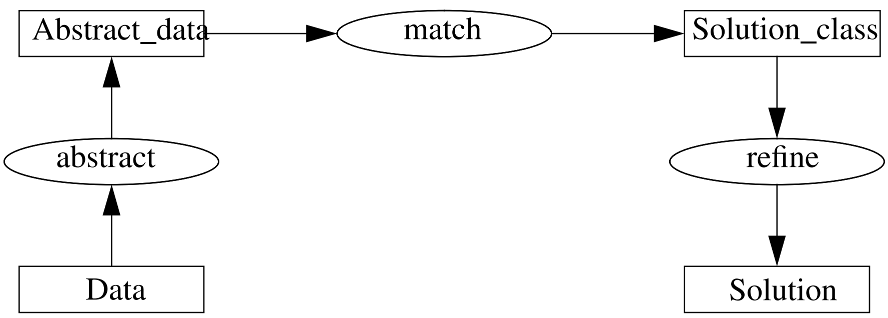


Figure 8. Inference structure of heuristic classification

A possible control flow for this problem-solving method can be defined by the following P-KARL program:

Abstract_data := abstract (Data);
Solution_class := match (Abstract_data);
Solution := refine (Solution_class)

The corresponding alphabet P of the P-KARL language is

P = ({abstract, match, refine}, {Abstract_data, Data, Solution, Solution_class}, ∅, ∅)

The alphabet L of a L-KARL language

$L = (E, \Sigma, \text{Boolean}, \text{Integer}, \text{Real}, \text{String}, A_{E\rightarrow}, A_{\Sigma\rightarrow}, A_{E\rightarrow\rightarrow}, A_{\Sigma\rightarrow\rightarrow}, \Pi, A_\Pi, X_E, X_\Sigma, X_V)$ with
Σ = {abstract_data, data, solution, solution_class}
Π = {abstract_relation, match_relation, refine_relation}
A_Π = {specific_, abstract_, diagnosis_, solution_}

is used to define a semantical structure for this program.

The ***carrier*** is the power set of the Herbrand base of L which is the infinite set of all ground formulae which can be constructed using the alphabet L. A set of class definitions CD_i is associated with every ***dataflow variable*** $x_i \in X$.

- $CD_{Abstract\_data}$ = {CLASS abstract_data ... END.}
- CD_{Data} = {CLASS data ... END.}
- $CD_{Solution}$ = {CLASS solution ... END.}
- $CD_{Solution\_class}$ = {CLASS solution_class ... END.}

Three sets of rules and definitions of L have to be defined to interpret the function symbols:

- The rule and definition set $S_{abstract}$:
  CLASS abstract_data ... END.[1]
  CLASS data ... END.
  PREDICATE abstract_relation
      specific_ : {};
      abstract_ : {}
  END.
  ∀x ∀y (y ε abstract_data ← x ε data ∧ abstract_relation(specific_ : x, abstract_ : y)).
  ∀x ∀y ... (abstract_relation(specific_ : x, abstract_ : y)) ← ...).
  ...
- The rule and definition set S_{match}:
  CLASS abstract_data ... END.
  CLASS solution_class ... END.
  PREDICATE match
      abstract_ : {};
      diagnosis_ : {}
  END.
  ∀x ∀y (y ε solution_class ← x ε abstract_data ∧ match(abstract_ : x, diagnosis_ : y)).

---

1. Because of well-typing, the class definitions of the dataflow variables must be repeated. In chapter three, the definition of an inference action is defined in a way that this is no longer necessary.

$\forall$x $\forall$y ... (match(abstract_ : x, diagnosis_ : y)) $\leftarrow$ ...).
...

- The rule and definition set S_{refine}:
CLASS solution ... END.
CLASS solution_class ... END.
PREDICATE refine
    diagnosis_ : {};
    solution_ : {}
END.
$\forall$x $\forall$y (y ε solution $\leftarrow$ x ε solution_class $\wedge$ refine(diagnosis_ : x, solution_ : y)).
$\forall$x $\forall$y ... (refine(diagnosis_ : x, solution_ : y)) $\leftarrow$ ...).
...

The semantics of the entire program

(Abstract_data := abstract (Data);
Solution_class := match (Abstract_data);
Solution := refine (Solution_class))$^A \subseteq \wp$(HB) x $\wp$(HB)

are all tuple of variable assignments (ν,μ) with ν(data) contains a set of input facts which describe the case to be classified and μ(solution) is a set of ground facts which contains the classification according to the given input facts.

Chapter 3

KARL at the Epistemological Level

A ***model of expertise*** according to the KADS-I project (cf. [BrW89], [WSB92]) separates different kinds of knowledge and provides different epistemological primitives for each of these types of knowledge.

- The ***domain layer*** contains domain-specific knowledge about concepts, their features, their elements, and their relationships. This knowledge is task-neutral in the sense that there are no means to control its use. The objects and the terminology of the domain which the knowledge-based system is to use are described at this layer.
- The ***inference layer*** contains knowledge about the used problem-solving method. This layer indicates which inferences are necessary within the problem-solving method. This knowledge describes the application of the domain knowledge for inferencing. It describes the logical inferences and their dependencies. No causal dependencies or causal ordering between these logical implications are given.
- The ***task layer*** contains knowledge about the control flow of a problem-solving method for solving a specific task. This layer specifies sequence, iteration, and alternatives of inferences.
- The ***strategic layer*** contains knowledge about the combination and selection of tasks for solving a problem adequately. This layer is not considered by KARL, but some ideas about a possible formalization are given.

In this chapter, it will be shown for every layer which language primitives are offered by KARL and how the different layers are connected by KARL. The so-called Sisyphus example[1] (cf. [Lin92c], [Lin92d]) will serve to illustrate each layer: The problem is to assign employees to available rooms by fulfilling given constraints. For example, it is not very useful

1. The Sisyphus example was used during several knowledge acquisition workshops to compare different modelling approaches.

to put a smoker and a non-smoker into the same room. A very simplified problem-solving method, i.e. beam search without backtracking, will be used to simplify the example. More elaborated problem-solving methods are given in [AFL91a], [AFL92b] and in chapter 5.

The model of expertise is extended by the so-called ***model of cooperation*** (cf. [GrB92]) to model the cooperation of the expert systems with other agents. Section 3.6 deals with the additional language constructs of KARL, which allow the representation of this cooperation.

3.0 The Sisyphus Example

"Sisyphus is a project that aims at comparing different approaches to aspects of knowledge engineering." (cf. [Lin92c], [Lin92d]) It defines an assignment problem, in which employees are assigned to office places with several requirements to be met.

At the domain layer there is a list of employees, their names and roles (like researcher), and their projects. It is also indicated if the employees smoke or not, if they are hackers or not, and who are their immediate co-workers. In addition, there is a list of rooms with a description of how they are situated with respect to the building and to the other rooms, and how many people they can accommodate.

The problem-solving process is described by a sample transcript of a protocol which describes the derivation of an assignment by an expert, the wizard Siggi D. Two conclusion are indicated in this protocol:

- In every step Siggi D. selects one employee using the following heuristics:
  - Place head of the group;
  - place his staff (secretaries and then managers), which should be next to the head of the group;
  - place heads of large groups, which should be next to the head of the group;
  - place others.
- Requirements for an assignment
  - Heads of large projects and the head of the group must get a single room;
  - researcher of the same project should not share a common room;
  - smokers and non-smokers should never be placed into the same room.

These two groups of knowledge are used to define the view of the problem-solving method at the domain layer. The first group of knowledge is used to select employees and places which are assigned next (cf. the views "Next_Comp" and "Next_Slot" in section 3.2). And the second group of knowledge is used to check whether an assignment is correct (cf. the view "Correct_State" in section 3.2).

In the following section 3.1 the domain-specific concepts of the problem are defined. Section 3.2. defines roles and single inference steps of the problem-solving process, and section 3.3. defines the control flow between them. Beam search without backtracking is used here as problem-solving method, thus it cannot be guaranteed that a solution will be found even if one exists.

The problem-solving method differs from the way the expert solves the problem due to the fact that it is not intended here to create a cognitively adequate model of her problem-solving behaviour but to model an agent which can solve the task effectively and efficiently. Therefore, her behaviour is generalized. Instead of regarding only one assignment, the problem-solving method investigates several assignments in parallel.

3.1 The Domain Layer

The domain layer is divided into two parts: a ***knowledge part*** and a ***data part***.

- ***Case-independent (static) domain knowledge*** which is modelled by the knowledge engineer, and expresses features and dependencies of objects, concepts and relationships which apply in the domain, irrespective of the current case. This type of knowledge does not change during problem-solving, i.e. this knowledge can neither be modified by the user, nor by the system during a consultation. Changing the knowledge part is only possible as a result of knowledge acquisition by the knowledge engineer or as a result of machine learning, not, however, as a result of creating intermediate data in the process of solving a current problem.
- ***Case-specific (dynamic) data*** which are specific to the current case. Case-specific data are either data which are input by the external agent, or results computed by the system during problem-solving. Therefore the system needs read access and write access to these data. Case-specific data are replaced at least partially if a new case is considered. Case-specific data only convey information which consists of factual knowledge, whereas intensional aspects of domain knowledge are expressed exclusively as static knowledge. The case-specific data part consists of an ***input-data part*** and an ***output-data part***. The data part is initialized with input data before the problem-solving starts, and it will contain the result of the system when the problem-solving process has finished.

The problem-solving process may read from the knowledge part and from the input-data part. It can write to the output-data part. As it can neither read from the output-data part nor write to the input-data part or knowledge part, the domain layer contains no intermediate results[1].

Additionally, the knowledge part of a domain layer is divided into two parts, see figure 9.

- ***factual, assertional*** or ***individual domain knowledge*** about entities: facts applying in the domain.
- ***general domain knowledge:*** terminological knowledge and knowledge about general features of domain entities. The terminological knowledge is modelled by class definitions and predicate definitions. The knowledge about general features contains intensional domain knowledge about entities (i.e., sufficient conditions), and necessary conditions (i.e., constraints).

In the following, language primitives for terminological knowledge, sufficient conditions (intensional descriptions), necessary conditions (constraints), factual knowledge, and data are introduced. The language primitives for the factual knowledge are also used for the data part. A domain layer is described by a tuple (^{D}L, ^{D}D, ^{D}R, ^{D}F, ^{D}C, i, o) which will be defined further on. Then an example demonstrating the derivation of new object IDs, the semantics of a

1. Otherwise, inference actions could communicate by means of the domain layer without sharing a common role. This kind of dirty programming or hidden inference structures can be prevented if the domain layer is not used as a global variable with read-and-write access of all inference actions. If all inference actions can only read from the knowledge part and the input-data part and can only write to the output-data part, their communication must be mediated by common stores, see section 3.2.

domain layer, the main theme of the domain layer, and the Sisyphus example are given.

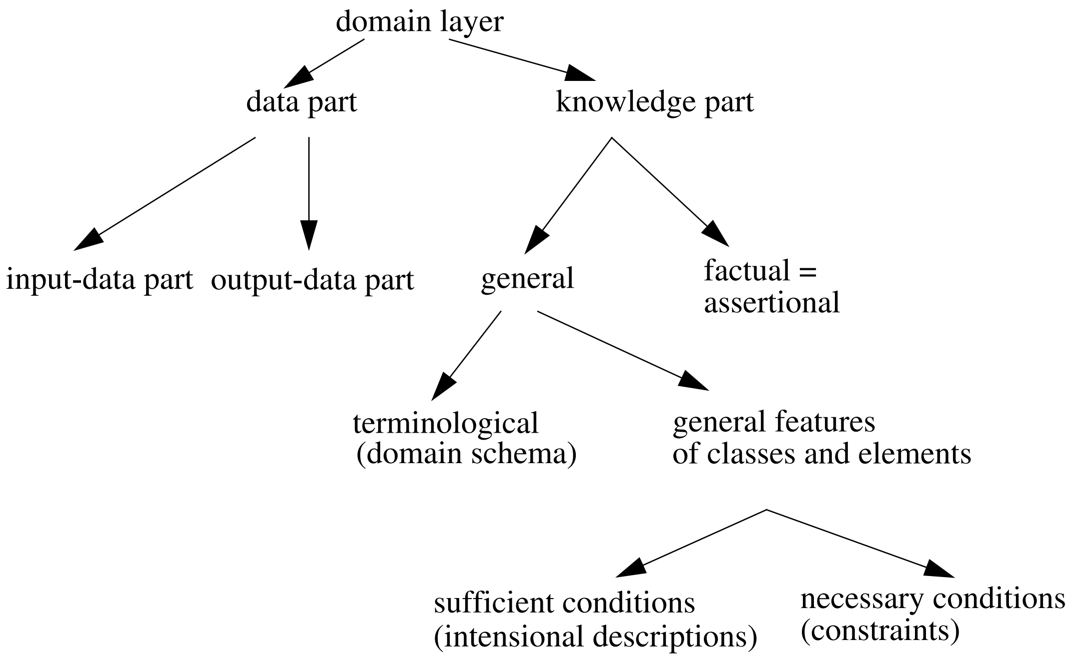


Figure 9. The structure of the domain layer

3.1.1 Terminological Knowledge: The Domain Schema

The ***schema of a domain layer***, i.e. the structure of the knowledge and data used at the domain layer, is defined by the means of the L-KARL language. It defines an alphabet ${}^{D}L$ and a set of definitions ${}^{D}D$ over ${}^{D}L$.

Def. 77 Alphabet of a domain layer

An ***alphabet ${}^{D}L$ of a domain layer*** is a tuple

$${}^{D}L = ({}^{D}E, {}^{D}\Sigma, \text{Boolean}, \text{Integer}, \text{Real}, \text{String}, {}^{D}A_{E\rightarrow}, {}^{D}A_{\Sigma\rightarrow}, {}^{D}A_{E\rightarrow\rightarrow}, {}^{D}A_{\Sigma\rightarrow\rightarrow}, {}^{D}\Pi, {}^{D}A_{\Pi}, X_E, X_\Sigma, X_V)$$

where

- ${}^{D}E$ defines the domain-specific element denotations,
- ${}^{D}\Sigma$ defines the domain-specific class names,
- Boolean defines the value set {False, True},
- Integer defines the value set of all integers,
- Real defines the value set of all real numbers,
- String defines the value set of all strings with finite length,

- ${}^{D}A_{E\rightarrow}$ defines the domain-specific single-valued (i.e., functional) attributes for elements,
- ${}^{D}A_{\Sigma\rightarrow}$ defines the domain-specific single-valued (i.e., functional) attributes for classes,
- ${}^{D}A_{E\rightarrow\rightarrow}$ defines the domain-specific set-valued attributes for elements,
- ${}^{D}A_{\Sigma\rightarrow\rightarrow}$ defines the domain-specific set-valued attributes for classes,
- ${}^{D}\Pi$ defines the domain-specific predicate symbols,
- ${}^{D}A_{\Pi}$ defines the domain-specific argument names of predicates,
- X_E defines the element variables,
- X_Σ defines the class variables,
- X_V defines the value variables.

Def. 78 Definitions of a domain layer

${}^{D}D$ denotes a set of class definitions and predicate definitions over the alphabet ${}^{D}L$ and is called the ***class definitions and predicate definitions of a domain layer***.

In the following it will be discussed how a ***concept***, a ***subclass relationship***, a ***part-of relationship***, and ***arbitrary relationships*** can be modelled at the domain layer.

Def. 79 Concept

A ***concept*** is represented by a class name $c \in {}^{D}\Sigma$ and a class definition for "c". The structure which is used to describe the concept and its instances, i.e. elements of the class "c", is defined by the class definition.

A class definition

CLASS c
    ISA c_1,..., c_n;
    CLASS_ATT
        ca_1 : { ca_{11},..., ca_{1k_1} };
        ...;
        ca_r : { ca_{r1},..., ca_{rk_r} };
        cs_1 :: { cs_{11},..., cs_{1l_1} };
        ...;
        cs_t :: { cs_{t1},..., cs_{tl_t} };
    ELEMENT_ATT
        ea_1 : { ea_{11},..., ea_{1m_1} };
        ...;
        ea_u : { ea_{u1},..., ea_{rm_u} };
        es_1 :: { es_{11},..., es_{1n_1} };
        ...;

$es_v :: \{es_{v1},..., es_{vn_v}\};$
END

defines the attributes which can be used for these descriptions and additionally the super classes of “c”.

Four different kinds of attributes exist:

- $ca_i \in {}^{D}A_{\Sigma\rightarrow}$ is a ***functional*** attribute which can be used to describe the class. The functionality of an attribute implies that “c” can only have one value for it.
- $cs_j \in {}^{D}A_{\Sigma\rightarrow\rightarrow}$ is a ***set-valued*** attribute which can be used to describe the class. “c” can have a set of values for it.
- $ea_i \in {}^{D}A_{E\rightarrow}$ is a ***functional*** attribute which can be used to describe elements of the class. The functionality of an attribute implies that an element can only have one value for it.
- $es_j \in {}^{D}A_{E\rightarrow\rightarrow}$ is a ***set-valued*** attribute which can be used to describe elements of the class. An element can have a set of values for it.

The definition of an attribute
$a : \{c_1,..., c_m\}$ or $a :: \{c_1,..., c_m\}$
also includes a range restriction. Every value of the attribute “a” must be an element or subclass of all classes c_i which are elements of its range restriction.

The row
ISA $c_1,..., c_n$;
has two consequences:

- Every element of “c” is also an element of c_i, i.e. the extension of “c” is a subset of the extension of c_i.
- c inherits all attributes which are defined by the class definition of c_i.

Def. 80 Subclass relationship

As discussed above, a ***subclass relationship*** between two concepts can be defined by the ISA row of the according class definition.

A ***part-of relationship*** or ***complex objects*** can be defined between elements of concepts “c” and “c´” defining single-valued or set-valued attributes for elements in the class definition of “c” having “c´” as range restriction. The two following class definitions illustrate this:

- CLASS c
  ELEMENT_ATT
  part : {c´};
  END
- CLASS c
  ELEMENT_ATT
  part :: {c´};
  END

In the first case, an element of "c" can have one element of "c´" as part, whereas in the second case, an element of "c" can have an arbitrary number of elements of "c´" as parts.

Def. 81 Arbitrary relationships

> Arbitrary ***relationships*** between concepts and their instances can be represented by a class definition containing the concepts as range restrictions of an attribute, or by a predicate definition containing the concepts as domain restrictions of the arguments.

For example:

```
CLASS married_couple
   ELEMENT_ATT
      husband :  {male};
      wife :     {female};
END
```
[1]

Note that every element of "married_couple" is regarded as an object. If the elements of a relation should not be regarded as objects by themselves the relationship can be modelled by a predicate definition.

```
PREDICATE married_couple
   husband :     {male};
   wife :        {female};
END
```

Section 3.1.7 will give an example which illustrates the difference between classes and predicates. For technical details, see section 2.1.

3.1.2 Intensional Descriptions

By using intensional descriptions it is possible to describe classes, the extension of classes, and the extension of predicates or parts of them.

In dealing with real-world objects of a domain one needs symbols, i.e. denotations, which provide a reference. The following defines:

- ***denotations for elements*** which are used as reference to real-world objects,
- ***denotations for classes (i.e., concepts)*** which are used as reference to sets of real-world elements with common properties (i.e., they can be described by the same attributes), and
- ***denotations for values*** which are used to describe features of real-world objects or concepts.

1. The example is a little bit conservative.

Def. 82 Denotation of elements, classes, and values

- E-ID terms can be used to ***denote elements***, i.e. individual objects. An E-ID term consists of functions and constants of ^{D}E and variables of X_E.
- C-ID terms can be used to ***denote classes***, i.e. concepts. A C-ID term is a constant of $^{D}\Sigma$ or a variable of X_Σ.
- V-ID terms can be used to ***denote values***. A V-ID term is a constant of Boolean $\cup$ Integer $\cup$ Real $\cup$ String or a variable of X_V.

(cf. Def. 2 "ID terms" at page 48).

The two following definitions define ***is-element-of relationships*** between elements and classes and ***is-a relationships*** (i.e., subset relationships between the elements of classes) between classes.

Def. 83 Is-element-of relationship

An ***is-element-of relationship*** between elements which are denoted by an E-ID term "e", and classes which are denoted by a C-ID term "c", is modelled by an is-element-of term:

$e \, \varepsilon \, c$

(cf. Def. 6 "F-term" at page 51).

Def. 84 Is-a relationship

An ***is-a relationship*** between classes which are denoted by an C-ID term "c", and classes which are denoted by a C-ID term "c´", is modelled by an is-a term:

$c \leq c'$

(cf. Def. 4 "Is-a term" at page 49)

The first difference between an is-a term and an ISA-row of a class definition is the fact that an is-a term need not be a ground term, i.e. it can be defined for class variables of X_Σ. The second difference is the fact that it can be used in rules as a premise. The following example shows how a non-ground is-a term can be used to collect all superclasses of a class as the value of a set-valued attribute:

$\forall x_\Sigma \, (c[\text{subclass\_of} :: \{x_\Sigma\}] \leftarrow c \leq x_\Sigma)$.

The next definition introduces ***equality relationships*** between denotations. If there is an equality relationship between (syntactically) different denotations they denote the same element, class, or value, i.e. they are semantically identical.

Def. 85 Equality relationship

An ***equality relationship*** between two ID-terms "o", "o´" is modelled by an equality term:

$o \cong o'$
(cf. Def. 6 “F-term” at page 51)

The problem of ***equality*** arises in the context of functional attributes. Let, for example, the two ground data terms be given:

o[a : v]
o[a : v´]

Two possibilities exist to deal with this situation.

- First, ***inconsistency*** can be assumed as the functionality of the single-valued attribute “a” is violated because two different values “v” and “v´” are given for the same object “o”.
- Second, the ***equality*** of “v” and “v´” ($v \cong v'$) is assumed, i.e. “v” and “v´” are syntactically different denotations of the same entity. Then, no inconsistency is given.

The following distinction has been introduced:

- Syntactically different ground E-ID terms like “Mary” and “mother(John)” can denote the same real-world object. They define different views or accesses on a real-world object. Therefore, **the unique-name assumption does *not* hold for element denotations**. As E contains not only constants but functions which can be used to construct new object denotations this seems to be a necessary assumption (cf. [KiW93, KLW90]).
- Syntactically different ground C-ID terms, i.e. class names which are constants, denote different real-world classes. Therefore, **the unique-name assumption holds for classes**. The reason for this is mainly that the terminological knowledge provided by the class (and predicate) definitions should be static. These definitions should completely define the class hierarchy and the attributes which are available for classes and their elements. If the derivation of equality of class denotations were allowed explicitly by a conclusion of a rule like
  $c \cong c' \leftarrow ...$
  or implicitly by functional attributes like[1]
  $\forall x_{\sum}\ (c[a : x_{\sum}] \leftarrow c \leq x_{\sum})$.
  the terminology (i.e., the class hierarchy and the defined attributes which are available for classes and their elements) could change in dependence of the given facts. Therefore, different case-specific input would possibly lead to different terminologies. This is prevented by adopting the unique-name assumption for class denotations. Implicit or explicit equality of class names is always indicated as an inconsistency, i.e. the according set of definitions and rules has no model.
- Syntactically different ground V-ID terms, i.e. values which are constants, denote different values. Therefore, **the unique-name assumption holds for values**. Examples like
  $2 \cong 3$.
  must be ruled out, because they might lead to nonsense.

1. The rule would identify all super-classes of “c” as the same class.

This should not be confused with the equality of non-ground ID terms. The equality term

$... \leftarrow ... \wedge c \cong x_{\Sigma} \wedge ...$

used in a rule "binds" the value of x_{Σ} on the constant "c", which is not an inconsistency.

The next definition introduces the ***possibility of describing elements and classes by the values of single-valued (i.e., functional) and set-valued attributes***.

Def. 86 Description of objects and concepts by attributes

- The ***description of a real-world object with an attribute value*** can be accomplished by a data term
  o[a : v]
  where "o" is an E-ID term, "a" is a single-valued attribute for elements, i.e. $a \in {}^{D}A_{E\rightarrow}$, and "v" is either a data term or a V-ID term. If "v" is a data term, the attribute describes a relationship between the object "o" and further objects, classes, and values according to the data term "v". If "v" is a V-ID term then the object is described by the value "v".
- The ***description of a concept with an attribute value*** can be accomplished by a data term:
  c[a : v]
  where "c" is a C-ID term, "a" is a single-valued attribute for classes, i.e. $a \in {}^{D}A_{\Sigma\rightarrow}$, and "v" is either a data term or a V-ID term. If "v" is a data term, the attribute describes a relationship between the concept "c" and further objects, classes, and values according to the data term "v". If "v" is a V-ID term the concept is described by the value to which "v" refers.
- The ***description of a real-world object with a set of attribute values*** can be accomplished by a data term:
  o[a :: {v}]
  where "o" is an E-ID term, "a" is a set-valued attribute for elements, i.e. $a \in {}^{D}A_{E\rightarrow\rightarrow}$, and "v" is either a data term or a V-ID term. If "v" is a data term, the attribute describes a relationship between the object "o" and further objects, classes, and values according to the data term "v". If "v" is a V-ID term the object is described by the set of values to which "v" refers.
- The ***description of a concept with a set of attribute values*** can be accomplished by a data term:
  c[a :: {v}]
  where "c" is a C-ID term, "a" is a set-valued attribute for classes, i.e. $a \in {}^{D}A_{\Sigma\rightarrow\rightarrow}$, and "v" is either a data term or a V-ID term. If "v" is a data term, the attribute describes a relationship between the concept "c" and further objects, classes, and values according to the data term "v". If "v" is a V-ID term the concept is described by the set of values to which "v" refers.
  (cf. Def. 6 "F-term" at page 51)

As data terms need not be flat, i.e. the value of an attribute applied in a data term can be again a data term, ***complex values*** can be modelled.

Def. 87 Complex values

> ***Complex values*** are described by non-flat data terms.

An example is:
o[component : o'[left : l, right : r]]

Arbitrary relationships between classes, elements and values can be described by P-terms. An element of the extension of a predicate, i.e. an element of the relation which is used to interpret it, is not an object with an ID of its own. Therefore it is not possible that a relation contains several tuples with the same values (the same ground ID terms) for all components. If a relation possibly contains several copies of the same tuple it must be modelled by a class. Then it becomes possible to distinguish identical tuples by different object identities, i.e. by different E-ID terms. In section 3.1.7, this topic is discussed in more detail.

Def. 88 Arbitrary relationships

> ***Arbitrary relationships*** between real-world objects, concepts, and values can be modelled by P-terms or data terms. Using a P-term implies that the corresponding relation does not contain several tuples with the same values (i.e., reference to the same object, concept, or value) for all components. If a relation possibly contains several copies of the same tuple it must be modelled by a data term.
> (Cf. Def. 8 "P-term" at page 52)

The next construct can be employed to express ***logical relationships*** between other expressions. A ***rule*** can be used to infer new knowledge from given knowledge. Rules help to express the following issues:

- The derivation of a new is-element-of relationship, if the premises of the rule are evaluated to true.
- The derivation of new equality relationships between E-ID terms, if the premises of the rule are evaluated to true.
- The derivation of new attribute values for a concept or an element, if the premises of the rule are evaluated to true.
- The derivation of new elements of a relation, if the premises of the rule are evaluated to true.

Def. 89 Logical relationships

> ***Logical relationships*** are expressed by rules.
>
> (cf. Def. 53 "Rule" at page 95)

As every single data term, is-element-of term, equality term (which are called F-terms, cf. Def. 6 "F-term" at page 51), and P-term is also a rule this can be summed up with the following definition.

Def. 90 Rule set of a domain layer

${}^{D}R$ denotes a set of rules over the alphabet ${}^{D}L$ and is called the ***rule set of a domain layer***.

3.1.3 Factual Knowledge

Def. 91 Factual knowledge

Factual knowledge is represented by

- ***ground is-element-of terms***,
- ***ground equality terms***,
- ***ground data terms***, and
- ***ground P-terms***

over the alphabet ${}^{D}L$.

Def. 92 The set of factual knowledge

The set of facts which represent the ***factual knowledge***, is denoted by ${}^{D}F$.

3.1.4 Necessary Descriptions

Necessary descriptions define integrity constraints for a domain layer. They are not used to infer new knowledge but to check the modelled knowledge for correctness. Such necessary conditions can be modelled by arbitrary formulae (cf. Def. 9 “Formulae” at page 52).

Def. 93 Necessary descriptions

The ***necessary descriptions*** at a domain layer are modelled by a set of constraints ${}^{D}C = ({}^{D}C_1, {}^{D}C_2)$ over the alphabet ${}^{D}L$.
(cf. Def. 61 “Set of necessary constraints” at page 100)

${}^{D}C$ is pair of formulae sets ${}^{D}C_1$ and ${}^{D}C_2$, i.e. ${}^{D}C = ({}^{D}C_1, {}^{D}C_2)$, which are treated differently. ${}^{D}C_1$ can be used to extend the set of derivable facts at the domain layer. These facts are not added to the domain layer, but ${}^{D}C_2$ checks the original facts and their extensions for correctness. If a domain layer models a set of edges of a graph, for example, then ${}^{D}C_1$ can be used to derive all paths which can be derived from the edges, and ${}^{D}C_2$ can be used to check whether this graph is free of cycles (cf. section 2.1.8).

3.1.5 Data (Input-Data Part and Output-Data Part)

Def. 94 Data

Case-specific data are represented by

- ***ground is-element-of terms***,
- ***ground equality terms***,
- ***ground data terms***, and
- ***ground P-terms***

over the alphabet of ${}^{D}L$.

The ***input-data part*** contains case-specific input data and a set ${}^{i}D$ of class definitions and predicate definitions with ${}^{D}D \models {}^{i}D$. ${}^{i}D$ defines the terminology which is used to describe these data. The ***output-data part*** contains results of a problem-solving process and a set ${}^{o}D$ of class definitions with ${}^{D}D \models {}^{o}D$. ${}^{o}D$ defines the terminology which is used to describe these data.

Def. 95 Input-data part

The input-data part “i” is a tuple $(|i|, {}^{i}D)$. $|i|$ is a set of facts which represent the ***case-specific input data***. It is a set of ground F-terms and P-terms. ${}^{i}D$ is a set of class definitions and predicate definitions which describe the terminology used to describe these data. It holds ${}^{D}D \models {}^{i}D$.[1]

Def. 96 Output-data part

The output-data part “o” is a a tuple $(|o|, {}^{o}D)$. $|o|$ is a set of facts which represent the ***case-specific output data***. It is a set of ground F-terms. ${}^{o}D$ is a set of class definitions which describe the terminology used to describe these data. It holds ${}^{D}D \models {}^{o}D$.

${}^{o}D$ is used in the context of an elementary inference action. It determines the subset of a model of an elementary inference action which is used to update the content $|o|$ of the output-data part (cf. Def. 110 “Output of an elementary inference action” at page 160).

1. ${}^{i}D$ could be used to define an editor for the input of case-specific data.

3.1.6 Summary: All Parts of a Domain Layer

A domain layer is described by a tuple (^DL, ^DD, ^DR, ^DF, ^DC, i, o):

- ^DL defines an alphabet for a domain layer.
- ^DD is a set of class definitions and predicate definitions.
- ^DR is a set of rules, i.e. sufficient conditions.
- ^DF is a set of facts, i.e. factual knowledge.
- $^DC = (^DC_1, {}^DC_2)$ is a set of constraints, i.e. necessary conditions ensuring consistency.
- $i = (|i|, {}^iD)$ is a set of facts and definitions representing case-specific input data.
- $o = (|o|, {}^oD)$ is a set of facts and definitions representing case-specific output data.

3.1.7 An Example Demonstrating how the Derivation of New Objects is Modelled in KARL

An example for intensional descriptions which shows how the derivation of new objects is modelled in KARL, will be given in the following passage (cf. [KiW89]).

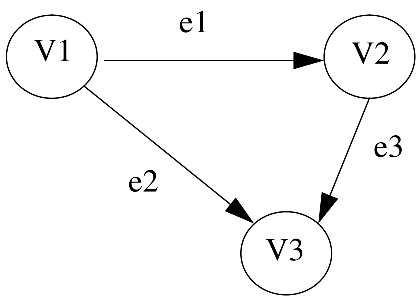
First, the terminological knowledge is defined by the following class definitions:

```
CLASS vertex
END;

CLASS edge
    ELEMENT_ATT
        start :     {vertex};
        end :       {vertex};
END;

CLASS path
    ELEMENT_ATT
        start :     {vertex};
        end :       {vertex};
END;

CLASS path´
    ELEMENT_ATT
        start :     {vertex};
        end :       {vertex};
END
```

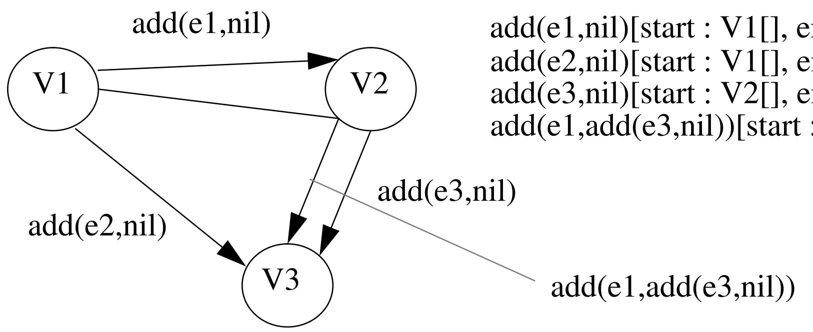


V1 ε vertex, V2 ε vertex, V3 ε vertex
e1[start : V1[], end : V2[]] ε edge,
e2[start : V1[], end : V3[]] ε edge,
e3[start : V2[], end : V3[]] ε edge

Figure 10. The elements of the classes "vertex" and "edge".

Then new objects are derived which describe all paths defined by elements of "edge". Note that, in this example, a new object is derived for every different path which connects two vertices. The ID of an element of "path" depends on all intermediate vertices and the ordering in which they are connected by an element of "path".

- The following rule (1) derives a new object for every existing element of "edge"; it defines the "start" and "end"-vertex of the object and specifies that these newly derived objects are elements of the class "path":
  (1) $\forall e_E \; \forall x_E \; \forall y_E$
  $(\text{add}(e_E,\text{nil})\ [\text{start} : x_E, \text{end} : y_E]\ \varepsilon\ \text{path} \leftarrow e_E[\text{start} : x_E, \text{end} : y_E]\ \varepsilon\ \text{edge})$.
- The following rule (2) derives new objects by appending an element of "edge" to an already existing element of "path" and specifies that these newly derived objects are elements of the class "path":
  (2) $\forall e_E \; \forall p_E \; \forall x_E \; \forall y_E \; \forall z_E$
  $(\text{add}(e_E,p_E)[\text{start} : x_E, \text{end} : y_E]\ \varepsilon\ \text{path} \leftarrow$
  $e_E[\text{start} : x_E, \text{end} : z_E]\ \varepsilon\ \text{edge} \wedge p_E[\text{start} : z_E, \text{end} : y_E]\ \varepsilon\ \text{path})$.



add(e1,nil)[start : V1[], end : V2[]] ε path,
add(e2,nil)[start : V1[], end : V3[]] ε path,
add(e3,nil)[start : V2[], end : V3[]] ε path,
add(e1,add(e3,nil))[start : V1[], end : V3[]] ε path

Figure 11. The elements of the class "path"

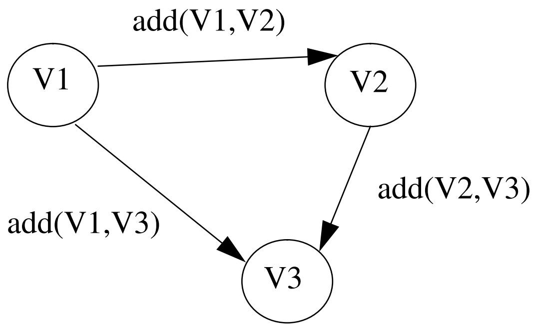
If it is, however, only a matter of interest whether one vertex is connected with a second one, the following rules apply. In this case, an element of "path´" is defined for every pair of vertices (i,j) which are connected, and not for every different path which connects two vertices (i,j) as in the definition of “path”. The ID of an element of “path´” only depends on the ID of its “start” and “end” vertix.

- The following rule (3) derives a new object for every existing element of “edge”; it defines the “start” and the “end”-vertex of the object and specifies that these newly derived objects are elements of the class “path´”:

  (3) $\forall e_E\ \forall x_E\ \forall y_E$
  $(add(x_E,y_E)[start : x_E,\ end : y_E]\ \varepsilon\ path' \leftarrow e_E[start : x_E,\ end : y_E]\ \varepsilon\ edge)$.

- The following rule (4) derives new objects by appending an element of “edge” to an already existing element of “path´” and specifies that these newly derived objects are elements of the class “path´”:

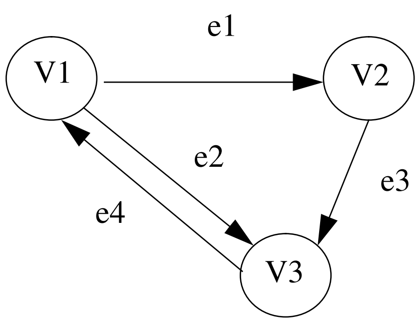
  (4) $\forall e_E\ \forall p_E\ \forall x_E\ \forall y_E\ \forall z_E$
  $(add(x_E,y_E)[start : x_E,\ end : y_E]\ \varepsilon\ path' \leftarrow$
  $e_E[start : x_E,\ end : z_E]\ \varepsilon\ edge \wedge p_E[start : z_E,\ end : y_E]\ \varepsilon\ path')$.



add(V1,V2)[start : V1[], end : V2[]] ε path´,
add(V1,V3)[start : V1[], end : V3[]] ε path´,
add(V2,V3)[start : V2[], end : V3[]] ε path´

Figure 12. The elements of the class “path´”

If a fourth edge e4[start : V3[], end : V1[]] is added as an element to the class “edge” the difference becomes even more important.



V1 ε vertex,
V2 ε vertex,
V3 ε vertex

e1[start : V1[], end : V2[]] ε edge,
e2[start : V1[], end : V3[]] ε edge,
e3[start : V2[], end : V3[]] ε edge,
e4[start : V3[], end : V1[]] ε edge,

Figure 13. The elements of the classes “vertex” and “edge” extended by “e4”

Then, “path” would have an infinite extension, i.e. it would have an infinite number of elements, whereas “path’” would have only six more elements.

In the following, this example is used to discuss the difference between data-terms and P-terms. P-terms (predicates) can be used to model relationships between objects. The tuples which are used to interpret a P-term are no objects by themselves. Therefore their identities are determined by their attribute values, i.e. by the values of their components. A predicate, i.e. the relationship modelled by it, cannot have two tuples with the same values for every component. Every element of "path´" has different values for the attributes "start" and "vertex", i.e. the identity term of an element add(x_E,y_E) depends functionally on the attribute values x_E and y_E, therefore a binary predicate "Path" can be used in similar way. ***This would not be possible for the class "path"***, which could contain elements with the same attribute values for "start" and "end" but with different ID-terms.

The following will give the predicate definition and the rules, which "compute" the extension of the predicate "Path" similar to the class "path´".

- PREDICATE Path
  start : {vertex};
  end : {vertex};
  END
- (5) $\forall e_E\ \forall x_E\ \forall y_E$
  (Path(start : x_E, end : y_E) $\leftarrow$ e_E[start : x_E, end : y_E] ε edge).
- (6) $\forall e_E\ \forall x_E\ \forall y_E\ \forall z_E$
  (Path(start : x_E, end : y_E) $\leftarrow$
  e_E[start : x_E, end : z_E] ε edge $\wedge$ Path(start : z_E, end : y_E)).

A last example to illustrate this difference:

If "married_couple"
CLASS married_couple
ELEMENT_ATT
husband : {male};
wife : {female};
END
is modelled as a class, it can contain the same married couple, i.e. objects which have the same values for "husband" and "wife", several times. This can make sense, if two persons get married several times. For example, the same couple can re-marry after a divorce. If it were established that "married_couple" contains the same combination of "husband" and "wife" only once, "married_couple" could be modelled by a binary predicate.

PREDICATE married_couple
husband : {male};
wife : {female};
END

(cf. section 3.1.1)

3.1.8 Semantics of the Domain Layer

The language which is applied at the domain layer is L-KARL. It is a syntactical modification of predicate logic, but it preserves the model-theoretic semantics of first-order logic. Therefore, the ***minimal-model semantics*** (cf. [Llo87], [Ull88]) or the ***perfect-model semantics*** (cf. [Prz88]) can be applied for the domain layer.

Let a domain layer (^{D}L, ^{D}D, ^{D}R, ^{D}F, ^{D}C, i, o) be given. The well-typed perfect Herbrand model (cf. Def. 59 “Well-typed perfect Herbrand model” at page 98) of

$^{D}D \cup {}^{D}R \cup {}^{D}F \cup |i|$

is defined as semantics of the ***input-data part and knowledge part of a domain layer***. The well-typed perfect Herbrand model (cf. Def. 59 “Well-typed perfect Herbrand model” at page 98) of

$^{D}D \cup {}^{D}R \cup {}^{D}F \cup |o|$

is defined as semantics of the ***output-data part and knowledge part of a domain layer***.

This semantics enforces that the rule set ^{D}R and the corresponding sets of facts ^{D}F and |i| or |o| are locally stratified (cf. [Prz88]). If ^{D}R is a rule set containing only Horn clauses this model is identical with the well-typed least Herbrand model (cf. Def. 51 “Well-typed least Herbrand model” at page 93). L-KARL offers three kinds of integrity conditions which must hold for this model.

A model, i.e. the untyped perfect Herbrand model, violates the ***equality constraints***, if it contains a ground equality term which indicates the equality of different class denotations or value denotations, or the equality of denotations with different types, for example, the equality of an element denotation and a value denotation. This error could be the result of explicitly given equality terms or the implicit result of functional attributes. If the untyped perfect Herbrand model violates the equality constraints, an ***equality error*** is indicated (cf. Def. 38 “Equality error” at page 81).

A model violates the ***well-typing conditions***. The well-typing conditions ensure that a model does not violate the domain restrictions and range restrictions for attributes and the domain restrictions for predicates (cf. section 2.1.4.2). An attribute can only be applied to an element or class if this is allowed by a corresponding class definition, and the value of an attribute must conform to all range restrictions of the attribute. Similarly, an element of the relation which interprets a predicate symbol must conform to the domain restriction of the predicate. Therefore, it is not the perfect Herbrand model that is immediately regarded here as semantics, but the untyped perfect Herbrand model. If the perfect Herbrand model were regarded as semantics, the model would contain all additional class definitions, predicate definitions, is-a terms, and is-element-of terms which ensure the well-typing conditions for all derivable data-terms and P-terms.[1] But what is really wanted here is that all derivable data-terms and P-terms meet the well-typing conditions according to the given class definitions and predicate definitions ^{D}D. Therefore, a two step “procedure” is applied here: First, the untyped perfect Herbrand model is regarded for which the well-typing condition need not hold. In the case of Horn clauses this is the untyped least Herbrand model. Then it is checked whether this untyped perfect Herbrand model is also a model, i.e. whether the well-typing conditions apply for it. If they do, it is called the well-typed perfect Herbrand model, and adopted as semantics. If they do not, a ***typing error*** is indicated.

1. This inflatoric semantics would “repair” all violated typing requirements by infering additional is-element-of terms, is-a terms, and class defnitions. This kind of abductive inference would hid all typing errors (cf. [KLW90]).

If the well-typed perfect Herbrand model exists and satisfies the set of constraints ${}^{D}C$, it is called the correct well-typed perfect Herbrand model (cf. Def. 62 “Correct well-typed perfect Herbrand model” at page 100). Otherwise, a ***violation of the constraints*** is indicated.

The following definition sums up these conditions. The input-data part is regarded here in combination with the knowledge part, so is the output-data part. The three parts of a domain layer do not have a semantics of their own, because the data parts refer to knowledge which is defined at the knowledge part, and the knowledge refers to the data which are defined in the data parts. The union of all three parts need not to have a semantics because the output data are results of the problem-solving process, which is described at the inference layer and task layer. The value of a single-valued attribute of an input data can be changed by the problem-solving process, for example. This change cannot be declaratively represented at the domain layer.

Def. 97 The correct well-typed perfect Herbrand model of the input-data part and knowledge part of a domain layer

Let a domain layer (${}^{D}L$, ${}^{D}D$, ${}^{D}R$, ${}^{D}F$, ${}^{D}C$, i, o) be given.

- If the untyped perfect Herbrand model of ${}^{D}D \cup {}^{D}R \cup {}^{D}F \cup |i|$ does not exist, an equality error is indicated.
- If the untyped perfect Herbrand model of ${}^{D}D \cup {}^{D}R \cup {}^{D}F \cup |i|$ exists, but is not a model of ${}^{D}D \cup {}^{D}R \cup {}^{D}F \cup |i|$, a typing error is indicated.
- If the untyped perfect Herbrand model of
  ${}^{D}D \cup {}^{D}R \cup {}^{D}F \cup |i|$
  exists and is also a model of ${}^{D}D \cup {}^{D}R \cup {}^{D}F \cup |i|$ but does not satisfy ${}^{D}C$, it is a well-typed perfect Herbrand model which violates the integrity constraints.
- If the well-typed perfect Herbrand model of
  ${}^{D}D \cup {}^{D}R \cup {}^{D}F \cup |i|$
  exists and satisfies ${}^{D}C$ then it is adopted as the semantics of ${}^{D}D \cup {}^{D}R \cup {}^{D}F \cup |i|$ and is called the ***correct well-typed perfect Herbrand model*** of the input-data part and knowledge part of a domain layer (${}^{D}L$, ${}^{D}D$, ${}^{D}R$, ${}^{D}F$, ${}^{D}C$, i, o).

Def. 98 The correct well-typed perfect Herbrand model of the output-data part and knowledge part a domain layer

Let a domain layer (${}^{D}L$, ${}^{D}D$, ${}^{D}R$, ${}^{D}F$, ${}^{D}C$, i, o) be given.

- If the untyped perfect Herbrand model of ${}^{D}D \cup {}^{D}R \cup {}^{D}F \cup |o|$ does not exist, an equality error is indicated.
- If the untyped perfect Herbrand model of ${}^{D}D \cup {}^{D}R \cup {}^{D}F \cup |o|$ exists, but is not a model of ${}^{D}D \cup {}^{D}R \cup {}^{D}F \cup |o|$, a typing error is indicated.
- If the untyped perfect Herbrand model of
  ${}^{D}D \cup {}^{D}R \cup {}^{D}F \cup |o|$
  exists and is also a model of ${}^{D}D \cup {}^{D}R \cup {}^{D}F \cup |o|$ but does not satisfy ${}^{D}C$, it is a well-typed perfect Herbrand model which violates the integrity constraints.

• If the well-typed perfect Herbrand model of

$^{D}D \cup {}^{D}R \cup {}^{D}F \cup |o|$

exists and satisfies ^{D}C then it is adopted as the semantics of $^{D}D \cup {}^{D}R \cup {}^{D}F \cup |o|$ and is called the ***correct well-typed perfect Herbrand model*** of the output-data part and knowledge part of a domain layer (^{D}L, ^{D}D, ^{D}R, ^{D}F, ^{D}C, i, o).

3.1.9 The Main Theme of the Domain Layer

The domain layer of KARL applies the L-KARL language as defined in section 2.1. The significant features of L-KARL are:

- ***Object-orientation***: Real-world entities are denoted by objects and concepts are described by classes. Both can be further described by data terms.
- ***Multiple inheritance and well-typing***: To describe objects and concepts, attributes can be defined by definitions. Definitions define domain restrictions and range restrictions for attributes. This is checked by the so-called well-typing condition. A model of a set of rules and definitions of an L-KARL language must be well-typed. Attributes are inherited by is-a and is-element-of terms. Multiple inheritance is possible.
- ***Functional and set-valued attributes***: The value of an attribute depends either functionally on the ID of an object (or a concept), or it can be a set of values. The value of an attribute can be either an object, a concept, or a value.
- ***Horn logic extended by stratified negation*** (cf. [Prz88]): Rules in L-KARL can express the derivation of new facts by given ones. As negation in rule bodies is allowed, L-KARL can express the derivation of new information from negative information.
- ***Model-theoretic semantics***: The semantics of a domain layer is the so-called ***perfect Herbrand model*** (cf. [KLW90], [Prz88]) of the set of rules and definitions. This model is checked for equality errors, typing errors, and constraints violation.

3.1.10 The Domain Layer of the Sisyphus Example

This section defines the domain layer of the Sisyphus example. A set of employees is assigned to office places, under several conditions. The employees are modelled by the class “employee” defining several attributes which can be applied to its elements. The subclasses “boss” and “big_boss” are defined. The available places are modelled by the class “place”. In the following, the terminological knowledge is presented in a graphical way by the graphical primitives which are defined in chapter 4.

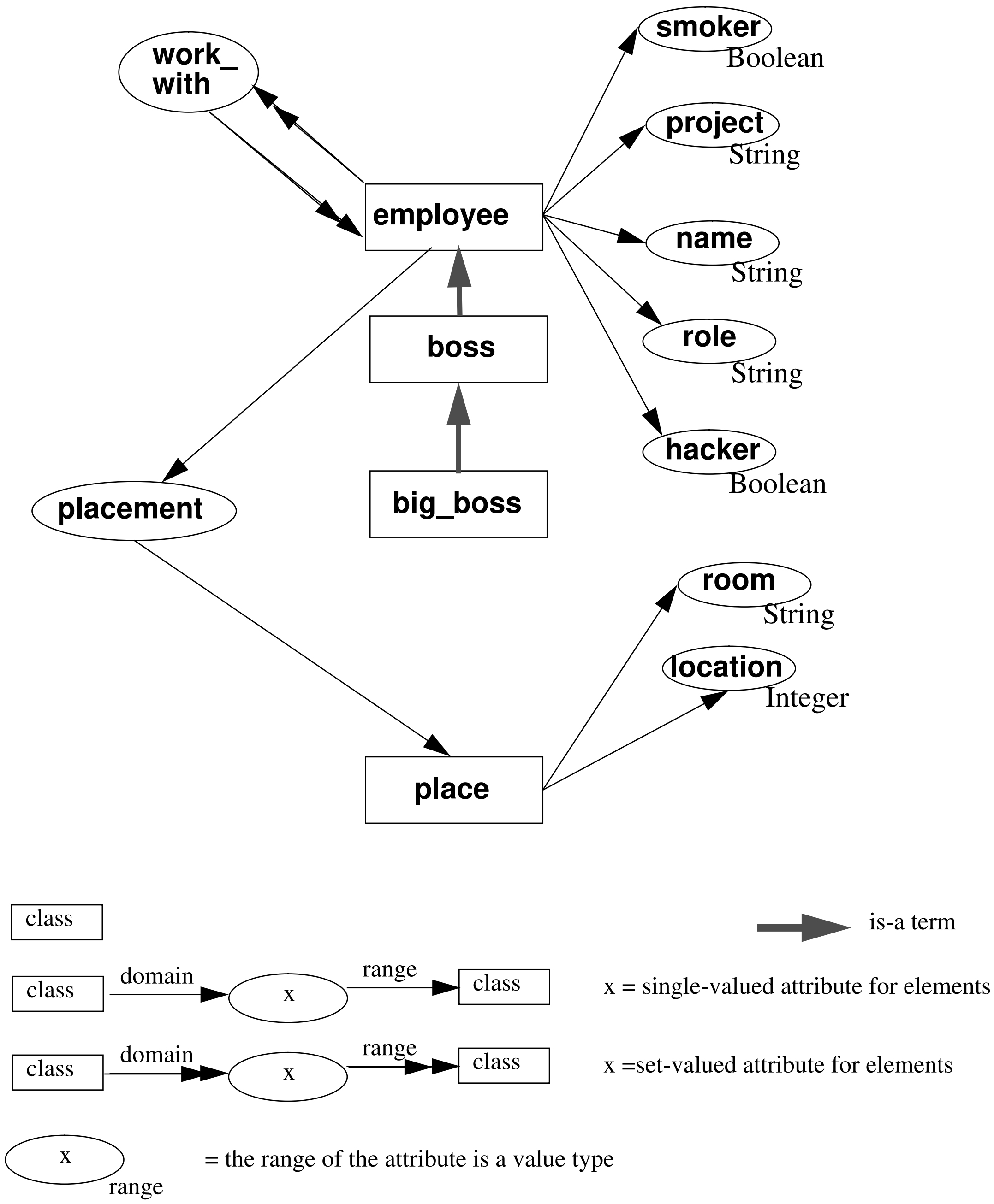


Figure 14. The domain schema of the Sisyphus example.

3.1.10.1 Domain schema: ${}^{D}L$ and ${}^{D}D$

Alphabet ${}^{D}L$[1]

ELEMENTS
${}^{D}E := \{\boldsymbol{wl}, \boldsymbol{jl}, \boldsymbol{td}, \boldsymbol{ei}, \boldsymbol{mx}, \boldsymbol{hw}, \boldsymbol{p0}, \boldsymbol{p1}, \boldsymbol{p2}, ...\}$

CLASSES
${}^{D}\Sigma := \{\boldsymbol{big\_boss}, \boldsymbol{boss}, \boldsymbol{employee}, \boldsymbol{place}\}$

EF_ATTRIBUTES
${}^{D}A_{E\rightarrow} := \{\boldsymbol{hacker}, \boldsymbol{location}, \boldsymbol{name}, \boldsymbol{placement}, \boldsymbol{project}, \boldsymbol{role}, \boldsymbol{room}, \boldsymbol{smoker}\}$

ES_ATTRIBUTES
${}^{D}A_{E\rightarrow\rightarrow} := \{\boldsymbol{work\_with}\}$

ELEMENT_VARIABLES
$X_E := \{x_{E1}, x_{E2}\}$

Definitions ${}^{D}D$

/* The class “employee” models the employees */
CLASS ***employee***
ELEMENT_ATT
hacker : {***Boolean***};
name : {***String***};
placement : {***place***};
project : {***String***};
role : {***String***};
smoker : {***Boolean***};
work_with :: {***employee***};
END;

/* The class “place” models the places in the rooms. The room containig a place is the value of the attribute “room”. The centrality of a place is indicated by the attribute “location”. */
CLASS ***place***
ELEMENT_ATT
location : {***Integer***};
room : {***String***};
END;

/* The class “big_boss” models the head of the group */
CLASS ***big_boss***
ISA ***boss;***
END;

/* The class “boss” models the heads of the projects. */
CLASS ***boss***
ISA ***employee;***

1. Only these parts of the alphabet are given which are not pre-defined (e.g., value types) or empty.

END;

3.1.10.2 Intensional Descriptions ^{D}R

This part is empty, i.e. no intensional knowledge is given.

3.1.10.3 Factual Knowledge ^{D}F

This part is empty, i.e. no factual knowledge is given.

3.1.10.4 Necessary Descriptions (Constraints) ^{D}C

$^{D}C_1$

$^{D}C_1$ is empty

$^{D}C_2$

% This constraint ensures that only one head of group exists. %

$\forall x_{E1}, \forall x_{E2}\ (x_{E1} \cong x_{E2} \leftarrow x_{E1}\ \varepsilon\ big\_boss \wedge x_{E2}\ \varepsilon\ big\_boss)$.

3.1.10.5 Input-Data Part "i"

^{i}D

The only difference between ^{D}D and ^{i}D is that the attribute "placement" for the class employee" is not defined here.

|i|:

class: employee

E-ID	hacker	name	project	role	smoker	work_ with
wl	True	"Werner"	"RESPECT"	"Researcher"	False	aw, mm
jl	True	"Jürgen"	"EULISP"	"Researcher"	False	hc, td
td	False	"Thomas"	"EULISP"	"Researcher"	False	jl, hb
ei	False	"Eva"		"Manager"	False	td, uu, mx
mx	False	"Monika"		"Secretary"	False	td, uu, ei

class: employee

E-ID	hacker	name	project	role	smoker	work_ with
hw	False	“Hans”	“BABYLON product”	“Researcher”	True	mt
...						

class: boss ≤ {employee}

E-ID	hacker	name	project	role	smoker	work_ with
hw	False	“Hans”	“BABYLON product”	“Researcher”	True	mt
...						

class: big_boss ≤ {boss, employee}

E-ID	hacker	name	project	role	smoker	work_ with
td	False	“Thomas”	“EULISP”	“Researcher”	False	mt

class: place

E-ID	location	room
p0	4	“c5-113”
p1	3	“c5-114”
p2	2	“c5-115”
p3	1	“c5-116”
p4	0	“c5-117”
p5	0	“c5-117”
p6	0	“c5-119”
p7	0	“c5-119”
p8	1	“c5-120”
p9	1	“c5-120”
...		

3.1.10.6 Output-Data Part "o"

^{o}D

CLASS ***employee***
  ELEMENT_ATT
    placement : {***place***};
END;

|o|

The output-data part is empty, because it will contain the results of the problem-solving process.

3.2 The Inference Layer

The inference layer describes ***generic*** inferences which use ***domain-specific*** knowledge. This is one of the significant characteristics of KADS, and this twin role is an important requirement in formalizing a model of expertise. These two kinds of knowledge must be separated clearly, i.e. for each kind of knowledge, specific language elements with clearly separated functions must be offered:

- On the one hand, the inference actions and the inference structure should only consist of ***generic*** knowledge.
- On the other hand, a view on ***domain-specific*** knowledge must be established, which allows the inference actions to use this knowledge.

If this can be achieved, reusability is possible in two ways: First, the knowledge about the problem-solving method is domain-independent and task-independent, i.e it may be used in several domains and for different tasks. Second, the same domain knowledge may be used for different problem-solving methods and different, though related tasks. Therefore, domain knowledge may be reused as well.

The inference layer defines a "dataflow point of view" (cf. [You89]) on the inferences drawn by an expert system.[1] It shows the single inference steps and their connections by shared premises and conclusions. The inference layer only indicates ***logical implications*** and how they are connected. No control flow and no causal relationships between the elements are given.[2]

The separation of the two aspects, i.e. the definition of a data flow between inference actions and the definition of domain views of them, is discussed in chapter 3.2.2.

The following will give a description of the language elements ***alphabet, role, elementary inference action***, ***inference structure***, and ***composed inference action***, which KARL provides at the inference layer. Finally, the inference layer of a solution for the Sisyphus problem is shown.

3.2.1 An Alphabet of an Inference Layer

An alphabet ^{I}L of an inference layer defines the names of the roles (i.e., the names of the views, stores, and terminators), the names of the elementary inference actions, and the names of the composed inference actions, and an alphabet of an L-KARL language.

1. "An inference structure can be viewed as a data-dependency diagram at the lowest level of functional decomposition." [Sch92a], p. 165.
2. A logical implication should not be confused with a causal relationship.

Def. 99 Alphabet of an inference layer

An ***alphabet ^IL of an inference layer*** is a tuple

$({}^IL, {}^IV, {}^IS, {}^IT, EIA, CIA)$

where

- ^IL defines an alphabet of an L-KARL language, i.e. it is a tuple

  $^IL = ({}^IE, {}^I\Sigma, Boolean, Integer, Real, String, {}^IA_{E\to}, {}^IA_{\Sigma\to}, {}^IA_{E\to\to}, {}^IA_{\Sigma\to\to}, {}^I\Pi, {}^IA_{\Pi}, X_E, X_\Sigma, X_V)$

  where

  - ^IE defines the generic element denotations,
  - $^I\Sigma$ defines the generic class names,
  - Boolean defines the value set {False, True},
  - Integer defines the value set of all integers,
  - Real defines the value set of all real numbers,
  - String defines the value set of all strings with finite length,
  - $^IA_{E\to}$ defines the generic single-valued (i.e., functional) attributes for elements,
  - $^IA_{\Sigma\to}$ defines the generic single-valued (i.e., functional) attributes for classes,
  - $^IA_{E\to\to}$ defines the generic set-valued attributes for elements,
  - $^IA_{\Sigma\to\to}$ defines the generic set-valued attributes for classes,
  - $^I\Pi$ defines the generic predicate symbols,
  - $^IA_{\Pi}$ defines the generic argument names of predicates,
  - X_E defines the element variables,
  - X_Σ defines the class variables,
  - X_V defines the value variables,
- ^IV is a finite set of cardinality v which defines the ***view*** names,
- ^IS is a finite set of cardinality s which defines the ***store*** names,
- ^IT is a finite set of cardinality t which defines the ***terminator*** names,
- EIA is a finite set of cardinality r which defines the names of the ***elementary inference actions***,
- CIA is a finite set of names which defines the names of the ***composed inference actions***.

$^DY \cap {}^IY = \emptyset$ if $Y \in \{E, \Sigma, A_{E\to}, A_{\Sigma\to}, A_{E\to\to}, A_{\Sigma\to\to}, \Pi, A_{\Pi}\}$. Therefore, disjunct sets of names are adopted for elements, classes, attributes, predicates, and argument names for the alphabets of the domain layer and the inference layer.

3.2.2 Roles

In the following it is shown that roles serve a dual purpose. They can be used to define a view between the inference layer and the domain layer, and they can be used to model the dataflow between inference actions. Owing to the fact that a view can be defined upward or downward, three different types of roles are identified: stores, views, and terminators.

3.2.2.1 A Typology of Roles

Roles have two types of tasks:

- The connection of domain layer and inference layer: Roles define a view from the inference layer to the domain layer by means of ***upward mappings***. The generic inference actions can use the domain knowledge by means of a role. Roles which are connected to the domain layer constitute a ***view*** of the problem-solving method onto the domain knowledge and case data - compare this discussion to the discussion on view definition in database design (cf. [BLN86], [ElN89]). Since a piece of domain knowledge may play various roles within a problem-solving method, several roles may be mapped onto the same expressions at the domain layer. In addition, a role can be used to map results of the inference layer, which are expressed in generic terms, back on domain layer expressions. The result of a problem-solving process, which is described by generic terms of the inference layer, is rephrased in domain-specific terms by means of a ***downward mapping*** of a role.
- The collection of intermediate data: These roles represent the dataflow dependency between inference actions. Such a role collects output of inference actions and provides intermediate results of the problem-solving process for inference actions. This type of role models the dataflow dependency between inference actions.

Therefore, there are two different dimensions for classifying roles:

- Roles define a ***connection between the domain layer and inference layer***. Four variations are possible:
  - No connection is defined.
  - An upward connection is defined, i.e. the role reads from the domain layer.
  - A downward connection is defined, i.e. the role writes to the domain layer.
  - A downward and an upward connection is defined, i.e. the role reads from and writes at the domain layer.
- Roles define ***the dataflow dependencies of inference actions*** at the inference layer, with three (four) possible variants:
  - A role serves as an input role of an inference action.
  - A role serves as an output role of an inference action.
  - A role serves as input roles and output roles of inference actions (Only this kind of role defines a data flow ***between*** inference actions).

• A role serves neither as input role nor as output role of an inference action. As this case does not make sense it is not regarded any further.

Combining both criteria, twelve different types of roles become visible.

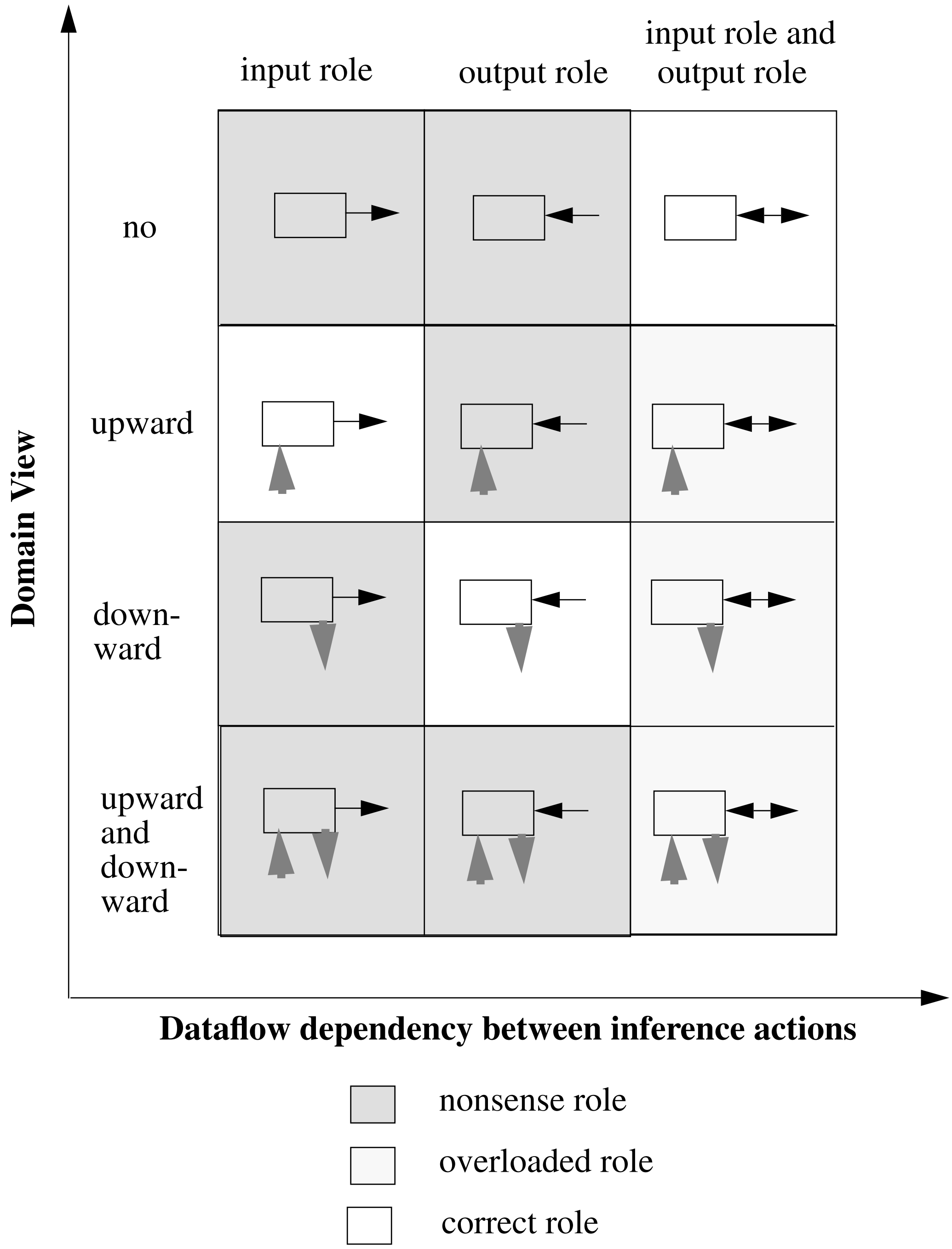


Figure 15. A typology of roles

Six of them do not make sense, like a role which is only an input role and has no connection with the domain layer, for example. Three of them are overloaded, i.e. they represent very complex interactions which should better be represented by different roles. They represent the data flow between inference actions because they are input roles **and** output roles. And they represent the connection of domain layer and inference layer. There remain three well-defined role types:

- ***Upward-view (Input role of an inference layer)***: An input role of an inference layer reads knowledge and data from the domain layer, which serve as input to an inference action (cf. the role “Correct_States“ of figure 18).[1] According to the terminology in the database domain, they will be referred to as ***views*** in the following. Views can be a means of reading from the knowledge part and the input-data part of a domain layer.
- ***Downward-view (Output role of an inference layer)***: An output role of an inference layer writes results of the problem-solving process back to the output-data part of the domain layer. The output role receives the results from an inference action which has used them as output role (cf. the role “Solution” of figure 18). As these roles obtain the final results of a problem-solving process, they are called ***terminators***. They can be used to write at the output-data part of a domain layer.

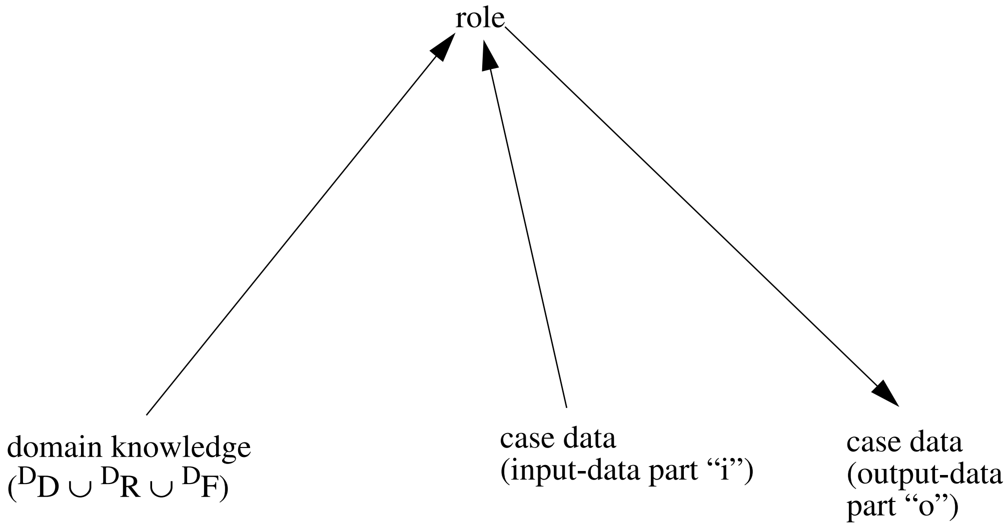


Figure 16. The different connections of domain layer and inference layer

- ***Dataflow dependency (Intermediate role of an inference layer)***: An intermediate role of an inference layer collects output from inference actions and serves as input to other inference actions. Such a role has no connection to the domain layer (cf. the roles “Possible_Solutions“, “States”, and “Successors” of figure 18). These roles store the result of an inference action. Other inference actions can use contents of these roles as an input. Such roles are called ***stores***.

3.2.2.2 Views

As the description of the problem-solving steps and their interactions should be generic, a view onto the domain knowledge must be defined. This view defines how domain-specific

1. This role type corresponds to what was called meta-relation in an older version of KARL (cf. [FAL91]).

knowledge and case-specific data is used for the inferences. These upward views are modelled by so-called ***views***. A view always serves to one or more inference actions as an input role. L-KARL is used here to describe a view. With every $view_i \in {}^{I}V$ a tuple $({}^{view_i}D, {}^{view_i}R)$ is associated.

Def. 100 View

A ***view*** $view_i$ is a tuple $({}^{view_i}D, {}^{view_i}R)$ where

- ${}^{view_i}D$ is a set of class definitions and predicate definitions over the alphabet ${}^{I}L$, and
- ${}^{view_i}R$ is a set of rules over the alphabet
$({}^{I}E \cup {}^{D}E, {}^{I}\Sigma \cup {}^{D}\Sigma$, Boolean, Integer, Real, String, ${}^{I}A_{E\rightarrow} \cup {}^{D}A_{E\rightarrow}, {}^{I}A_{\Sigma\rightarrow} \cup {}^{D}A_{\Sigma\rightarrow}, {}^{I}A_{E\rightarrow\rightarrow} \cup {}^{D}A_{E\rightarrow\rightarrow}, {}^{I}A_{\Sigma\rightarrow\rightarrow} \cup {}^{D}A_{\Sigma\rightarrow\rightarrow}, {}^{I}\Pi \cup {}^{D}\Pi, {}^{I}A_{\Pi} \cup {}^{D}A_{\Pi}, X_E, X_{\Sigma}, X_V)$.

The set ${}^{view_i}D$ of class definitions and predicate definitions is used to define classes and predicates which are used in the rules ${}^{view_i}R$. In the context of an elementary inference action applying the view, this set ${}^{view_i}D$ of class definitions and predicate definitions are used to define well-typing conditions for the model of an elementary inference action (cf. section 3.2.3.2).

The ***upward connection*** from the input-data part "i" and the domain knowledge ${}^{D}D \cup {}^{D}R \cup {}^{D}F$ to a view $view_i$ ***transforms*** domain-specific terminology and case-specific data into the generic terminology of the inference layer, and makes it identifiable to an elementary inference action applying the view. The connection of a view $view_i$ to the set of definitions ${}^{D}D$, rules ${}^{D}R$, facts ${}^{D}F$, and case-specific input data "i" of the domain layer is specified by the set of rules ${}^{view_i}R$. The alphabet for these rules can be obtained from the union of the alphabet of the inference layer and the alphabet of the domain layer because these rules set up connections between expressions over both alphabets.

The syntactical structure of a view $view_i$ is as follows:

```
VIEW view_i
        DEFINITIONS
              ...;
        UPWARD MAPPING
              ...;
END
```

3.2.2.3 Stores

Stores can be used to model the data flow, i.e. the input and output dependencies of inference actions. ***Input stores*** (premise roles) provide input to an inference action whereas results produced by a inference action are collected in ***output stores*** (conclusion roles). Input stores model premises of an inference step, output stores model the conclusions of an inference step.

Therefore, a ***store*** $\mathrm{store}_i \in {}^{I}S$ is modelled as a kind of ***container*** which can be filled with domain knowledge and case-specific data. In addition, intermediate results of the problem-solving process can be stored. L-KARL is used to describe a store. With every store_i a tuple $({}^{\mathrm{store}_i}D)$ is associated.

Def. 101 Store

A ***store*** store_i is a tuple $({}^{\mathrm{store}_i}D)$ where ${}^{\mathrm{store}_i}D$ is a set of class definitions over the alphabet ${}^{I}L$.

The set ${}^{\mathrm{store}_i}D$ of class definitions is applied to determine the contents of a store store_i and to define well-typing conditions for the contents of the store_i. The well-typing condition is proved in the context of an elementary inference action which has used the store as an input store or as an output store. Sets of facts are only allowed as contents of a store if they fit these definitions. The following definitions introduce the ***contents of a store***.

Def. 102 Contents of a store

The current ***contents*** $|\mathrm{store}_i|$***of a store*** store_i is a set of

- ground F-terms
- and the set ${}^{\mathrm{store}_i}D$.

The syntactical structure of a store store_i is as follows:

```
STORE store_i
        DEFINITIONS
                ...;
END
```

3.2.2.4 Terminators

As the description of the problem-solving steps and their interaction is generic, a downward view of them at the domain knowledge must be defined. This downward view defines how the results of a problem-solving process are expressed in domain-specific terms. Otherwise, the results would only be expressed in generic terms. These downward views are modelled here by so-called ***terminators***. A terminator is always used by one or more inference actions as a conclusion role. L-KARL is applied here to describe a terminator. With every $terminator_i \in {}^IT$ a tuple (${}^{terminator_i}D$, ${}^{terminator_i}R$) is associated.

Def. 103 Terminator

A ***terminator*** $terminator_i$ is a tuple (${}^{terminator_i}D$, ${}^{terminator_i}R$) where

- ${}^{terminator_i}D$ is a set of class definitions and predicate definitions over the alphabet ${}^IL$, and
- ${}^{terminator_i}R$ is a set of rules over the alphabet
($^IE \cup {}^DE$, $^I\Sigma \cup {}^D\Sigma$, Boolean, Integer, Real, String, ${}^IA_{E\rightarrow} \cup {}^DA_{E\rightarrow}$, ${}^IA_{\Sigma\rightarrow} \cup {}^DA_{\Sigma\rightarrow}$, ${}^IA_{E\rightarrow\rightarrow} \cup {}^DA_{E\rightarrow\rightarrow}$, ${}^IA_{\Sigma\rightarrow\rightarrow} \cup {}^DA_{\Sigma\rightarrow\rightarrow}$, ${}^I\Pi \cup {}^D\Pi$, ${}^IA_\Pi \cup {}^DA_\Pi$, X_E, X_Σ, X_V).

The set ${}^{terminator_i}D$ of class definitions and predicate definitions is used to define classes and predicates which are used as terms in the rules ${}^{terminator_i}R$. In the context of an elementary inference action applying the $terminator_i$ these class definitions and predicate definitions are used to define well-typing conditions for the result of an elementary inference action (cf. section 3.2.3.2).

The ***downward connection*** from the inference layer to the output-data part “o” ***transforms*** the generic terms of the inference layer into domain-specific terminology and case-specific data. The connection of a terminator $terminator_i$ to the output-data part “o” of the domain layer is specified by the set of rules ${}^{terminator_i}R$. The alphabet for these rules can be obtained from the union of the alphabets of the inference layer and of the domain layer because these rules set up connections between expressions over both alphabets.

The syntactical structure of a terminator $terminator_i$ is as follows:

```
TERMINATOR terminator_i
        DEFINITIONS
              ...;
        DOWNWARD MAPPING
              ...;
END
```

3.2.2.5 Roles

Def. 104 Role

A ***role*** "role" is either a store, a terminator, or a view:

role $\in {}^{I}S \cup {}^{I}T \cup {}^{I}V$

Roles denotes the set of roles, i.e. Roles = ${}^{I}S \cup {}^{I}T \cup {}^{I}V$.

Stores, terminators, and views (i.e., roles) do not have a semantics. Their task is to define a semantics of an elementary inference action which applyies them as premises or conclusions (cf. section 3.2.3.2).

3.2.3 Elementary Inference Actions

In the following, syntax and semantics of elementary inference actions are defined.

3.2.3.1 Description of an Elementary Inference Action

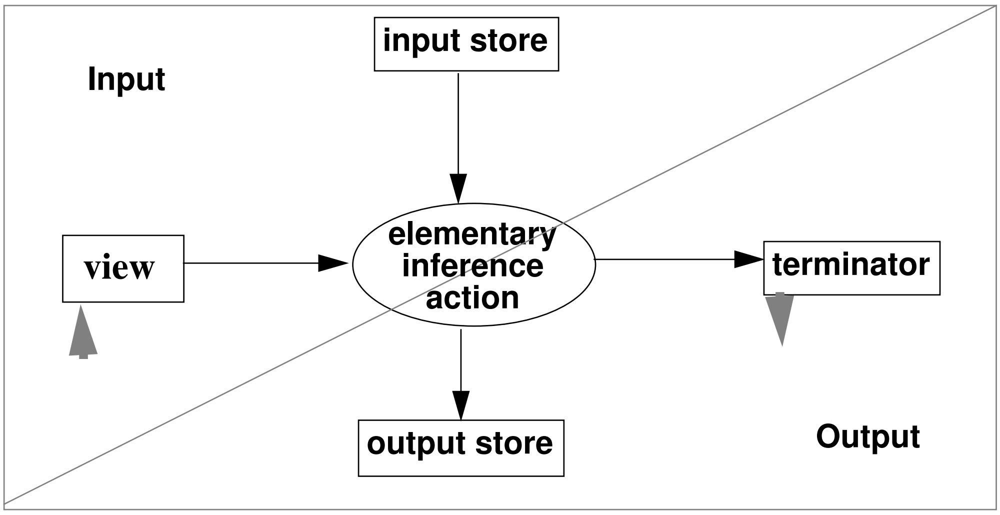


Figure 17. An elementary inference action and its input and output.

„A knowledge source (KS) is a functional description of an inference making process" [BrW89], p. 279. A knowledge source or elementary inference action $eia_j \in EAI$ has zero or more input roles (premise roles) and one or more output roles (conclusion roles). L-KARL is applied here to describe an elementary inference action. The logical derivation which is accomplished by an elementary inference action is modelled by a set ${}^{eia_j}R$ of rules over the alphabet ${}^{I}L$. With every elementary inference action eia_j a tuple (${}^{eia_j}\mathtt{Views}$, ${}^{eia_j}\mathtt{InputStores}$,

${}^{eia_j}\text{Terminators}$, ${}^{eia_j}\text{OutputStores}$, ${}^{eia_j}\text{D}$, ${}^{eia_j}\text{R}$, ${}^{eia_j}\text{C}$) is associated.

Def. 105 Elementary inference action

An ***elementary inference action eia$_j$*** is a tuple (${}^{eia_j}\text{Views}$, ${}^{eia_j}\text{InputStores}$, ${}^{eia_j}\text{Terminators}$, ${}^{eia_j}\text{OutputStores}$, ${}^{eia_j}\text{D}$, ${}^{eia_j}\text{R}$, ${}^{eia_j}\text{C}$) where

- ${}^{eia_j}\text{Views} \subseteq {}^{I}\text{V}$ is a possibly empty set of view names and defines views which are premises of eia$_j$,
- ${}^{eia_j}\text{InputStores} \subseteq {}^{I}\text{S}$ is a possibly empty set of store names and defines input stores (i.e., premises) of eia$_j$,
- ${}^{eia_j}\text{Terminators} \subseteq {}^{I}\text{T}$ is a possibly empty set of terminators names and defines terminators which are conclusions of eia$_j$,
- ${}^{eia_j}\text{OutputStores} \subseteq {}^{I}\text{S}$ is a possibly empty set of store names and defines output stores (i.e., conclusions) of eia$_j$,
- ${}^{eia_j}\text{D}$ is a set of class definitions and predicate definitions over the alphabet ${}^{I}\text{L}$,
- ${}^{eia_j}\text{R}$ is a set of rules over the alphabet ${}^{I}\text{L}$,
- ${}^{eia_j}\text{C}$ is a pair of two (possibly empty) sets of constraints (${}^{eia_j}\text{C}_1$, ${}^{eia_j}\text{C}_2$) over the alphabet ${}^{I}\text{L}$.

${}^{eia_j}\text{Views}$ defines the views, and ${}^{eia_j}\text{InputStores}$ defines the input stores of the elementary inference action eia$_j$. The union of both sets models the premises of eia$_j$. ${}^{eia_j}\text{Terminators}$ defines the (output-) terminators and ${}^{eia_j}\text{OutputStores}$ defines the output stores of eia$_j$. The union of both sets models the conclusions of this elementary inference action eia$_j$.

In addition to the definitions of the input roles and output roles, the set ${}^{eia_j}\text{D}$ of class definitions and predicate definitions is applied to define classes and predicates which are used by the rules ${}^{eia_j}\text{R}$. ${}^{eia_j}\text{R}$ describes the inference which is drawn by the elementary inference action. The set of constraints ${}^{eia_j}\text{C}$ defines integrity conditions for the model of an elementary inference action.

The syntactical structure of an elementary inference action eia$_j$ is as follows:

ELEMENTARY INFERENCE ACTION eia$_j$
    PREMISES
        ...;
    CONCLUSIONS
        ...;
    DEFINITIONS

```
        ...;
    RULES
        ...;
    NECESSARY CONDITIONS
        ...;
END
```

3.2.3.2 Semantics of an Elementary Inference Action

The language applied here to describe an elementary inference actions is L-KARL. It is a syntactical modification of predicate logic, but it preserves the model-theoretic semantics of first-order logic. Therefore, it is possible to apply the ***minimal-model semantics*** (cf. [Llo87], [Ull88]) or the ***perfect-model semantics*** (cf. [Prz88]) as a declarative semantics for an elementary inference action. The perfect-model semantics has been modified by the extensions discussed above (cf. section 2.1.7 and section 3.1.8):

- The model is checked for ***equality errors*** (cf. Def. 38 “Equality error” at page 81).
- The model is checked for ***typing errors*** (cf. section 2.1.4.2).
- The model is checked for violation of ***constraints*** (cf. section 2.1.8).

The following definitions define an ***input,*** a ***rule set,*** and a ***terminology*** of an elementary inference action. *Input* denotes the union of all contents of the stores which serve as input stores to the elementary inference action. *Rule set* denotes the union of all rules of terminators and views of the elementary inference action and the rules defined by the elementary inference action. If it has at least one view, this set also includes all rules which are defined at the domain layer. *Terminology* of an elementary inference action denotes the union of all class definitions and predicate definitions of input stores, output stores, views, and terminators being part of this inference action. In addition, *terminology* contains all class definitions and predicate definitions of the elementary inference action and all definitions from the domain layer if the elementary inference action has at least one terminator or view.

Def. 106 Input of an elementary inference action

Let an elementary inference action $eia_j = ({}^{eia_j}\mathrm{Views},\ {}^{eia_j}\overline{\mathrm{InputStores}},\ {}^{eia_j}\mathrm{Terminators},\ {}^{eia_j}\overline{\mathrm{OutputStores}},\ {}^{eia_j}D,\ {}^{eia_j}R,\ {}^{eia_j}C)$ be given.

If ${}^{eia_j}\overline{\mathrm{InputStores}} = \varnothing$ then $\overline{\mathrm{Input}}_{eia_j} := \varnothing$ else $\mathrm{Input}_{eia_j} := \bigcup_{store_i \in {}^{eia_j}\mathrm{InputStores}} |store_i|$

$\overline{\mathrm{Input}}_{eia_j}$ defines an ***input*** of an elementary inference action eia_j. It depends on the current contents of its input stores.

Def. 107 Rule set of an elementary inference action

Let an elementary inference action $eia_j = ({}^{eia_j}Views,\ {}^{eia_j}\bar{InputStores},\ {}^{eia_j}Terminators,\ {}^{eia_j}\bar{OutputStores},\ {}^{eia_j}D,\ {}^{eia_j}R,\ {}^{eia_j}C)$ be given.

- If ${}^{eia_j}Views = \emptyset$
then $Vrule_{eia_j} := \emptyset$ else $Vrule_{eia_j} := \left(\langle \bigcup_{view_i \in {}^{eia_j}Views} {}^{view_i}R\rangle \cup {}^{D}R \cup {}^{D}F \cup |i|\right)^1$

- If ${}^{eia_j}Terminators = \emptyset$
then $Trule_{eia_j} := \emptyset$ else $Trule_{eia_j} := \langle \bigcup_{terminator_i \in {}^{eia_j}Terminators} {}^{terminator_i}R\rangle$

- $Rule_{eia_j} := {}^{eia_j}R \cup Vrule_{eia_j} \cup Trule_{eia_j}$ defines the ***rule set*** of the elementary inference action eia_j.

Def. 108 Terminology of an elementary inference action

Let an elementary inference action $eia_j = ({}^{eia_j}Views,\ {}^{eia_j}\bar{InputStores},\ {}^{eia_j}Terminators,\ {}^{eia_j}\bar{OutputStores},\ {}^{eia_j}D,\ {}^{eia_j}R,\ {}^{eia_j}C)$ be given.

- If ${}^{eia_j}\bar{InputStores} = \emptyset$
then $Iterm_{eia_j} := \emptyset$ else $Iterm_{eia_j} := \bigcup_{store_i \in {}^{eia_j}InputStores} {}^{store_i}D$

- If ${}^{eia_j}Views = \emptyset$ then
$Vterm_{eia_j} := \emptyset$ else $Vterm_{eia_j} := \left(\left(\bigcup_{view_i \in {}^{eia_j}Views} {}^{view_i}D\right) \cup {}^{D}D\right)$

- If ${}^{eia_j}Terminators = \emptyset$ then
$Tterm_{eia_j} := \emptyset$ else $Tterm_{eia_j} := \left(\left(\bigcup_{terminator_i \in {}^{eia_j}Terminators} {}^{terminator_i}D\right) \cup {}^{D}D\right)$

- If ${}^{eia_j}\bar{OutputStores} = \emptyset$ then
$Oterm_{eia_j} := \emptyset$ else $Oterm_{eia_j} := \bigcup_{store_i \in {}^{eia_j}OutputStores} {}^{store_i}D$

1. "i" denotes the input-data part of the domain layer.

- $\mathrm{Term}_{eia_j} := {}^{eia_j}D \cup \mathrm{Iterm}_{eia_j} \cup \mathrm{Vterm}_{eia_j} \cup \mathrm{Tterm}_{eia_j} \cup \mathrm{Oterm}_{eia_j}$ defines the ***terminology*** of the elementary inference action eia_j.

It is now possible to define the meaning, i.e. the semantics, of an elementary inference action.

Def. 109 Correct well-typed perfect Herbrand model of an elementary inference action

Let an elementary inference action $eia_j = ({}^{eia_j}\mathrm{Views},\ {}^{eia_j}\overline{\mathrm{InputStores}},\ {}^{eia_j}\mathrm{Terminators},\ {}^{eia_j}\overline{\mathrm{OutputStores}},\ {}^{eia_j}D,\ {}^{eia_j}R,\ {}^{eia_j}C)$ and its input $\overline{\mathrm{Input}}_{eia_j}$ be given.

- If the untyped perfect Herbrand model of
  $\mathrm{Term}_{eia_j} \cup \mathrm{Rule}_{eia_j} \cup \overline{\mathrm{Input}}_{eia_j}$
  does not exist, an equality error is indicated.
- If the untyped perfect Herbrand model of
  $\mathrm{Term}_{eia_j} \cup \mathrm{Rule}_{eia_j} \cup \overline{\mathrm{Input}}_{eia_j}$
  exists but is no model of it, a typing error is indicated.
- If the untyped perfect Herbrand model of
  $\mathrm{Term}_{eia_j} \cup \mathrm{Rule}_{eia_j} \cup \overline{\mathrm{Input}}_{eia_j}$
  exists and is also a model of $\mathrm{Term}_{eia_j} \cup \mathrm{Rule}_{eia_j} \cup \overline{\mathrm{Input}}_{eia_j}$ but does not satisfy ${}^{eia_j}C$, then the elementary inference action has a well-typed perfect Herbrand model (cf. Def. 59 “Well-typed perfect Herbrand model” at page 98) which violates the integrity constraints (cf. Def. 61 “Set of necessary constraints” at page 100).
- If the well-typed perfect Herbrand model of
  $\mathrm{Term}_{eia_j} \cup \mathrm{Rule}_{eia_j} \cup \overline{\mathrm{Input}}_{eia_j}$
  exists and satisfies ${}^{eia_j}C$ then it is adopted as the semantics of the elementary inference action eia_j and is called its ***correct well-typed perfect Herbrand model*** (cf. Def. 62 “Correct well-typed perfect Herbrand model” at page 100).

Until now, the meaning of an elementary inference action has been defined in dependence of the current contents of its input stores. It remains now to define the modification of the contents of the output stores and the output data part which will be caused by the “execution” of an elementary inference action. The ***output*** of an elementary inference action modifies the contents of its output stores, and possibly the output-data part of the domain layer. Subsets of the correct well-typed perfect Herbrand model of the elementary inference action define the new contents of the output stores or the output-data part “o”. The set of all F-terms of the model of the elementary inference action is employed as the new contents of an output store or the output-data part ***which are terms over the class definitions*** of the store, or of the output-data part, respectively (cf. Def. 75 “Terms over a set of class definitions” at page 115).

Def. 110 Output of an elementary inference action

Let an elementary inference action $eia_j = ({}^{eia_j}\mathrm{Views},\ {}^{eia_j}\mathrm{Input\bar{S}tores},\ {}^{eia_j}\mathrm{Terminators},\ {}^{eia_j}\mathrm{Output\bar{S}tores},\ {}^{eia_j}D,\ {}^{eia_j}R,\ {}^{eia_j}C)$ and its current correct well-typed perfect Herbrand model H according to its current input be given. The ***output*** of an elementary inference action eia_j is the modification of the contents of its output stores and the output-data part "o" of the domain layer given below.

- If the elementary inference action has the store $store_i$ as conclusion, i.e. $store_i \in {}^{eia_j}\mathrm{Output\bar{S}tores}$, then the contents $|store_i|$ is updated by the set of all terms over the definitions ${}^{store_i}D$, i.e. $|store_i| := {}^{store_i}D(H)$.
- If the elementary inference action has a terminator as conclusion, i.e. ${}^{eia_j}\mathrm{Terminators} \neq \emptyset$, then $|o|$ is updated by the union of the old contents $|o|$ and the set of all terms over the definitions ${}^{o}D$, i.e. ${}^{o}D(H)$.[1]

An example which illustrates these abstract definitions is given in section 3.4.

3.2.4 Inference Structure

KARL extends the inference structures of KADS by hierarchical refinement as known from dataflow diagrams (cf. [You89]). Therefore, the following proposition does not hold for inference structures in KARL: "An inference structure can be viewed as a data-dependency diagram ***at the lowest level*** of functional decomposition." [Sch92a], p. 165. Instead, regarding inference structures in KARL, it is correct to say that an inference structure can be viewed as a data-dependency diagram.

3.2.4.1 Inference Structure

The complete inference structure is described by the premise roles and conclusion roles of the single inference actions. A store may be a conclusion role for one inference action and a premise role for another inference action. A view can be a premise for several inference actions, and a terminator can be a conclusion for several inference actions. In this way the whole graph structure of the inference structure is given. Note that ***no control knowledge or causal dependencies concerning the ordering of the inference actions*** are represented at the inference layer. Neither the sequence of definitions of inference actions, nor the sequence of clauses within the description of an elementary inference action implies an order of execution. Only logical inferences or logical dependencies are modelled.

Def. 111 Inference structure

An inference structure is a ***directed graph*** $IS = (IS_V, IS_E)$

1. Cf. Def. 96 "Output-data part" at page 134.

• The ***vertices*** of IS_V are all roles $role_i \in$ Roles, elementary inference actions $eia_j \in$ EAI, and composed inference actions $cia_k \in$ CAI of an inference layer.

• IS_E contains an ***edge*** from a role to an inference action if the role is an input role of the inference action.

• IS_E contains an ***edge*** from an inference action to a role if the role is an output role of the inference action.

3.2.4.2 Hierarchically Refined Inference Structures (his): Composed Inference Actions

Hierarchical refinement is a well-known technique of representing complex information on several layered levels. In the domain of software engineering, levelled dataflow diagrams (cf. [You89]) are a means to represent information on different levels of abstraction. The following reasons speak for this technique:

- Modelling complex systems within one inference structure will result in an unreadable diagram.
- It is possible to define several views onto a system or a problem-solving method. Every view shows the system at a different level of refinement.
- The top levels of inference structures which are developed bottom-up can be reused for similar tasks (cf. [LHS92]).
- If a system requires the cooperation of several predefined problem-solving methods, any of these methods can be represented by a composed inference action in one inference structure. The representation of the interactions of the methods is not mixed with their internal structure. For example, in [KFG92] the second level shows the interaction of three problem-solving methods that are all refined independently on additional levels.

KARL allows hierarchically levelled inference structures by introducing the language primitive ***composed inference action*** $cia_j \in$ CAI, that correspond to sub-nets of the whole inference structure. Composed inference actions allow a stepwise refinement and a hierarchical ordering of inference structures. A composed inference action is a set of elementary inference actions, composed inference actions, and their premise roles and conclusion roles. The hierarchical refinement of inference structures has the following properties:

- Only inference actions and not roles can be composed and therefore become refined.
- A composed inference action can be refined by other elementary and composed inference actions. They are called ***internal*** inference actions. The other inference actions are called ***external*** inference actions. Therefore, an inference action is either an external or an internal inference action according to a given composed inference action.
- A composed inference action must contain all input roles and output roles of its inference actions (i.e., its internal inference actions) which refine the composed inference action. They can be either internal or external ones. ***Internal roles*** are roles which are not used by other external inference actions but only by internal ones. ***External roles*** are roles

which are also used by other external inference actions. They are the input roles and output roles of the composed inference actions. If such a role only operates as an input role for some internal inference actions, then it is an input role of the composed inference action. If such a role only operates as an output role for some internal inference actions, then it is an output role of the composed inference action. Otherwise it is an input role and output role.

- The refinement of a composed inference action stops if it contains only elementary inference actions. When are inference actions elementary? ***An inference action is elementary if its behavior can be described declaratively without any control knowledge concerning the order of activities.*** This kind of control knowledge is described at the task layer by sequences of calls of inference actions. If an inference step represented by an inference action contains control knowledge which should be represented at the model of expertise, the inference action has to be broken up. If an inference step represented by a inference action does not contain any control knowledge there is no need to refine it.

In KARL an elementary inference action is declaratively represented by L-KARL expressions. This set defines the semantics of an elementary inference action. A composed inference action is described by its premise roles and conclusion roles, and by the set of inference actions and roles refining it. The semantics is determined by the declarative descriptions of their elementary inference actions and by the control flow between them as defined at the task layer. Therefore, the semantics of composed inference actions is discussed at the task layer.

In the following, composed inference actions are formally introduced, and constraints for a correct hierarchical refinement of inference structures by composed inference actions are defined. With every composed inference action cia_j a tuple (^{cia_j}Views, ^{cia_j}Input_Stores, ^{cia_j}Terminators, ^{cia_j}Output_Stores, ^{cia_j}InternalViews, ^{cia_j}Internal_Stores, ^{cai_j}InternalTerminators, ^{cai_j}InferenceActions) is associated.

Def. 112 Composed inference action

A ***composed inference action*** cia_j a tuple (^{cia_j}Views, ^{cia_j}Input_Stores, ^{cia_j}Terminators, ^{cia_j}Output_Stores, ^{cia_j}InternalViews, ^{cia_j}Internal_Stores, ^{cai_j}InternalTerminators, ^{cai_j}InferenceActions) where

- ^{cia_j}Views is a possibly empty set of role names and defines the ***input roles***, i.e. the premises, of cia_j which are views,
- ^{cia_j}Input_Stores is a possibly empty set of role names and defines the ***input roles***, i.e. the premises, of cia_j which are stores,
- ^{cia_j}Terminators is a possibly empty set of role names and defines the ***output roles***, i.e. the conclusions, of cia_j which are terminators,
- ^{cia_j}Output_Stores is a possibly empty set of role names and defines the ***output roles***, i.e. the conclusions, of cia_j which are stores,

• ${}^{cia_j}\mathrm{InternalViews}$ is a possibly empty set of role names and defines the ***internal roles*** of cia_j which are views,

• ${}^{cia_j}\mathrm{Int\bar{e}rnalStores}$ is a possibly empty set of role names and defines the ***internal roles*** of cia_j which are stores,

• ${}^{cai_j}\mathrm{InternalTerminators}$ is a possibly empty set of role names and defines the ***internal roles*** of cia_j which are terminators,

• ${}^{cia_j}\mathrm{InternalViews} \cup {}^{cia_j}\mathrm{Int\bar{e}rnalStores} \cup {}^{cai_j}\mathrm{InternalTerminators}$ is a **non-**empty set of role names,

• ${}^{cai_j}\mathrm{InferenceActions}$ is a non-empty set of names of elementary and composed inference actions and defines the ***internal inference actions*** of cia_j.

The following shows how the set ${}^{cai_j}\mathrm{InferenceActions}$ defines a substructure $IS_{cia_j} = (IS_{V_{cia_j}}, IS_{E_{cia_j}})$ of an inference structure IS = (IS_V, IS_E). In addition it is shown how the sets ${}^{cia_j}\mathrm{Views}$, ${}^{cia_j}\mathrm{Inpu\bar{t}Stores}$, ${}^{cia_j}\mathrm{Terminators}$, ${}^{cia_j}\mathrm{Outpu\bar{t}Stores}$, ${}^{cia_j}\mathrm{InternalViews}$, ${}^{cia_j}\mathrm{Int\bar{e}rnalStores}$, and ${}^{cai_j}\mathrm{InternalTerminators}$ depend on the set ${}^{cai_j}\mathrm{InferenceActions}$, i.e. how the set of internal inference actions determines the sets of premises, internal roles, and conclusions of a composed inference action.

Def. 113 Substructure of an inference structure

A composed inference action $cia_j \in$ CAI defines a ***substructure*** $IS_{cia_j} = (IS_{V_{cia_j}}, IS_{E_{cia_j}})$ ***of an inference structure*** IS = (IS_V, IS_E).

• The vertices of $IS_{V_{cia_j}}$ are all inference actions (i.e., $eia_i \in$ EAI and $cia_i \in$ CAI) which are defined as refinement of cia_j, i.e. which are elements of ${}^{cai_j}\mathrm{InferenceActions}$.

• In addition, $IS_{V_{cia_j}}$ contains a $role_i \in$ Roles if $role_i$ is a premise or conclusion of a vertice of $IS_{V_{cia_j}}$.

• $IS_{E_{cia_j}}$ contains an ***edge*** from a role to an inference action which is a vertex of $IS_{V_{cia_j}}$ if the role is a premise of the inference action.

• $IS_{E_{cia_j}}$ contains an ***edge*** from an inference action which is a vertex of $IS_{V_{cia_j}}$ to a role if the role is a conclusion of the inference action.

Such a substructure contains all internal inference actions, all internal roles, and all premise and conclusion roles of the composed inference actions and all edges which connect these elements. This substructure is pictured if a composed inference action is refined (cf. section 4.2 and chapter 5). The next definition introduces requirements for the premises, internal roles, and conclusions of a well-defined composed inference action.

Def. 114 Roles of a well-defined composed inference action

Let a composed inference action $cia_j \in CAI$, (${}^{cia_j}Views$, ${}^{cia_j}InputStores$, ${}^{cia_j}Terminators$, ${}^{cia_j}OutputStores$, ${}^{cia_j}InternalViews$, ${}^{cia_j}InternalStores$, ${}^{cai_j}InternalTerminators$, ${}^{cai_j}InferenceActions$) be given:

- cia_j has a $role_i \in$ Roles as ***premise,*** i.e. $role_i \in {}^{cia_j}Views \cup {}^{cia_j}InputStores$, if and only if there is an inference action as element of ${}^{cai_j}InferenceActions$ which has the role $role_i$ as premise and there is an inference action as vertex of $IS_V \setminus (IS_{V_{cia_j}} \cup \{cia_j\})$ which has the role $role_i$ as premise or conclusion.
- cia_j has a $role_i \in$ Roles as ***conclusion,*** i.e. $role_i \in {}^{cia_j}Terminators \cup {}^{cia_j}OutputStores$, if and only if there is an inference action as element of ${}^{cai_j}InferenceActions$ which has the role $role_i$ as conclusion and there is an inference action as vertex of $IS_V \setminus (IS_{V_{cia_j}} \cup \{cia_j\})$ which has the role $role_i$ as premise or conclusion.
- cia_j has a $role_i \in$ Roles as ***internal role***, i.e. $role_i \in {}^{cia_j}InternalViews \cup {}^{cia_j}InternalStores \cup {}^{cai_j}InternalTerminators$, if and only if there is an inference action as element of ${}^{cai_j}InferenceActions$ which has the role $role_i$ as premise or conclusion and there is ***no*** inference action as vertex of $IS_V \setminus (IS_{V_{cia_j}} \cup \{cia_j\})$ which has the role $role_i$ as premise or conclusion.

A well-defined composed inference action has a role as a premise (or as a conclusions, respectively) if an inference action which refines the composed inference action also has the role as a premise (or as a conclusions, respectively), and if an inference action which does not refine the composed inference action refers also to this role. A well-defined composed inference action has a role as internal role if an inference action which refines the composed inference action refers to this role, and if no inference action which does not refine the composed inference action refers to this role. In chapter four (cf. figure 21) some wrong refinements are illustrated.

The last two definitions are recursive because the roles of a composed inference action cia_j depend on the roles of other composed inference actions which refine cia_j. A check whether a refinement is correct could start with composed inference actions which are refined exclusively by elementary inference actions, and then proceed to the next level (i.e., stratification could be defined for inference actions). Currently, there is no tool support, i.e. the knowledge engineer has to enforce the correct refinement manually. Correct hierarchical refinement of graph structures are also discussed in the domain of Petri-nets (cf. [Rei90]).

The syntactical structure of a composed inference action cia_j is as follows:

```
COMPOSED INFERENCE ACTION cia_j
    PREMISES
        ...;
    CONCLUSIONS
        ...;
    ELEMENTARY INFERENCE ACTIONS
        ...;
    COMPOSED INFERENCE ACTIONS
        ...;
    VIEWS
        ...;
    STORES
        ...;
    TERMINATORS
        ...;
END
```

3.2.4.3 The Top-Level Inference Action

The so-called top-level inference action is a composed inference action which describes the complete problem-solving method by one composed inference action. Therefore it does not have any external roles, i.e. no premises and conclusions.

3.2.5 The Inference Layer of the Sisyphus Example

The example of an inference layer which is shown next introduces the inferences, their data flow dependencies and the role of the domain knowledge which are used to solve the Sisyphus task (cf. section 3.0). As several alternative assignments are regarded parallel, an assignment of several components (i.e., employees) and slots (i.e., places) is called a state. A state is a set of tuples of a component assigned to a slot. Five single inference steps are defined here which work with these states:

- ***Check***: This inference checks whether a state is a possible solution, i.e. whether all employees are placed by it. The input of "Check" are the elements of "States", and its output are the elements of "Possible_Solutions".

- ***Create***: This inference creates the successor states of the given states by assigning a new component to alternative slots. Its input are states (i.e., the store "States") and its output are the successors (i.e., the store "Successors"). It uses the domain views "Next_Comp" and "Next_Slots" which define the heuristics described in section 3.0.
- ***Init***: This inference creates the initial states. Its output are states (i.e., the store "States"), and it uses the domain views "Next_Comp" and "Next_Slots" which define the heuristics described in section 3.0.
- ***Prune***: This inference checks whether an assignment is correct. For this purpose it uses the domain view "Correct_State". It checks whether the requirements for a correct assignment are fulfilled by the state (cf. section 3.0). The input of "Prune" are the elements of "Successors" and the output are new elements of "States".
- ***Select_Solution***: This inference action selects one solution out of the derived possible solution. As all possible solutions place the employees with the highest priority in as centrally located rooms as possible, there is no additional criterion which can be used to select an "optimal" solution. Therefore, one of the possible solutions is selected at random and re-phrased in domain-specific terms by the terminator "Solution".

3.2.5.1 Alphabet

To avoid redundancy, the alphabet will not be given an explicit treatment here, it will, however, be dealt with in implicit definitions.

3.2.5.2 The Top-Level Inference Action "Assign"

/* The composed inference action "Assign" takes a set of components and a set of slots and computes sets out of sets of assignments. An assignment is a pair consisting of a component and a slot. A state is a set of assignments. "Assign" derives states which fit the given criteria for goal states. One goal state is selected as solution. */

TOP LEVEL INFERENCE ACTION ***Assign***
    ELEMENTARY INFERENCE ACTIONS
        Check, Create, Init, Prune, Select_Solution;
    VIEWS
        Correct_States, Next_Comp, Next_Slots;
    STORES
        PossibleSolutions, States, Successors;
    TERMINATORS
        Solution;
END

3.2.5.3 Refinement of "Assign"

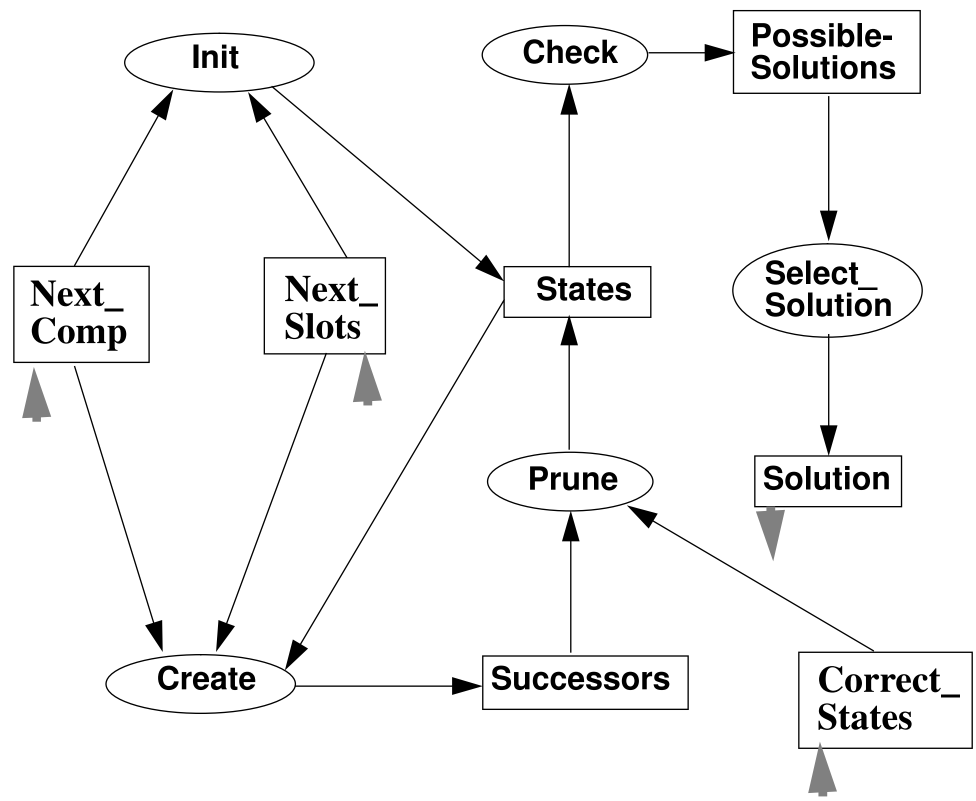


Figure 18. Refinement of "Assign"

3.2.5.3.1 Internal Stores

STORE ***PossibleSolutions***
  DEFINITIONS
    CLASS ***assignments***
      ELEMENT_ATT
        c : {***components***};
        s : {***slots***};
    END;
    CLASS ***components***
    END;
    CLASS ***possible_solutions***
      ELEMENT_ATT
        assign :: {***assignments***};
    END;
    CLASS ***slots***
    END;
END;

```
STORE States
   DEFINITIONS
      CLASS assignments
         ELEMENT_ATT
            c :      {components};
            s :      {slots};
      END;
      CLASS components
      END;
      CLASS slots
      END;
      CLASS states
         ELEMENT_ATT
            assign :: {assignments};
      END;
END;
```

```
STORE Successors
   DEFINITIONS
      CLASS assignments
         ELEMENT_ATT
            c :      {components};
            s :      {slots};
      END;
      CLASS components
      END;
      CLASS slots
      END;
      CLASS successors
         ELEMENT_ATT
            assign :: {assignments};
      END;
END;
```

3.2.5.3.2 Internal Terminators

TERMINATOR ***Solution***
  DOWNWARD MAPPING
    /* Every assignment of the solution is stored as the value of the attribute placement which is defined for employees. */
    $\forall x_A, \forall x_C, \forall x_S, \forall x_{St}$
      $(x_C[placement : x_S] \in employees \leftarrow$
        $x_{St}[assign :: \{x_A[c : x_C, s : x_S]\}] \in solution)$.
END;

3.2.5.3.3 Internal Domain Views

VIEW ***Corrcect_States***

DEFINITIONS

CLASS ***correct_states***

ISA ***successors;***

END;

CLASS ***wrong_states***

ISA ***successors;***

END;

PREDICATE ***fit_together***

c_1 : {***employee***};

c_2 : {***employee***};

END;

UPWARD MAPPING

/* Every state which assigns two employees who do not fit together to the same room is a wrong state */

$\forall x_A, \forall x_C, \forall x_N, \forall x_S, \forall x_{St}, \forall y_A, \forall y_C, \forall y_S$

$(x_{St} \,\varepsilon\, \mathit{wrong\_states} \leftarrow$

$x_{St}[\mathit{assign} :: \{x_A[c : x_C,\ s : x_S]\}] \,\varepsilon\, \mathit{successors} \wedge$

$x_{St}[\mathit{assign} :: \{y_A[c : y_C,\ s : y_S]\}] \,\varepsilon\, \mathit{successors} \wedge$

$\neg(x_C \cong y_C) \wedge$

$x_S[\mathit{room} : x_N] \,\varepsilon\, \mathit{place} \wedge$

$y_S[\mathit{room} : x_N] \,\varepsilon\, \mathit{place} \wedge$

$\neg\, \mathit{fit\_together}\ (c_1 : x_C,\ c_2 : y_C)).$

/* Two employees may share a room (1) if the roles of both are alike, (2) if either both smoke or neither of them does, (3) if neither of the two is a boss, and (4) if both work in different projects. */

$\forall x_C \forall x_{V1}, \forall x_{V2}, \forall x_{V3}, , \forall y_C, \forall y_V$

$(\mathit{fit\_together}\ (c_1 : x_C,\ c_2 : y_C) \leftarrow$

$x_C[\mathit{project} : x_{V1},\ \mathit{role} : x_{V2},\ \mathit{smoker} : x_{V3}] \,\varepsilon\, \mathit{employees} \wedge$

$y_C[\mathit{project} : y_V,\ \mathit{role} : x_{V2},\ \mathit{smoker} : x_{V3}] \,\varepsilon\, \mathit{employees} \wedge$

$\neg(x_C \,\varepsilon\, \mathit{boss}) \wedge$

$\neg(y_C \,\varepsilon\, \mathit{boss}) \wedge$

$\neg(x_{V1} \cong x_{V2}).$

/* Every successor state which is not a wrong state is a correct state */

$\forall x_{St}\ (x_{St} \,\varepsilon\, \mathit{correct\_states} \leftarrow x_{St} \,\varepsilon\, \mathit{successors} \wedge \neg(x_{St} \,\varepsilon\, \mathit{wrong\_states})).$

END;

VIEW ***Next_Comp***

DEFINITIONS

PREDICATE ***before_comp***
 high : {***components***};
 less : {***components***};
END;
UPWARD MAPPING
/* These rules define a preference ordering on the employees, that is head of group (i.e., big_boss), secretary, manager, head of project (i.e., boss), researcher. */

$\forall x_C, \forall y_C$
 $(before\_comp(high : x_C, less : y_C) \leftarrow$
 $x_C \, \varepsilon \, big\_boss \wedge y_C \, \varepsilon \, employees \wedge \neg(x_C \cong y_C)).$

$\forall x_C, \forall y_C, \forall y_V$
 $(before\_comp(high : x_C, less : y_C) \leftarrow$
 $x_C[role : \text{“Secretary”}] \, \varepsilon \, employees \wedge$
 $y_C[role : y_V] \, \varepsilon \, employees \wedge$
 $\neg(y_V \cong \text{“Secretary”}) \wedge$
 $\neg(y_C \, \varepsilon \, big\_boss)).$

$\forall x_C, \forall y_C, \forall y_V$
 $(before\_comp(high : x_C, less : y_C) \leftarrow$
 $x_C[role : \text{“Manager”}] \, \varepsilon \, employees \wedge$
 $y_C[role : y_V] \, \varepsilon \, employees \wedge$
 $\neg(y_V \cong \text{“Secretary”}) \wedge$
 $\neg(y_V \cong \text{“Manager”}) \wedge$
 $\neg(y_C \, \varepsilon \, big\_boss)).$

$\forall x_C, \forall y_C, \forall y_V$
 $(before\_comp(high : x_C, less : y_C) \leftarrow$
 $y_C[role : y_V] \, \varepsilon \, employees \wedge$
 $x_C \, \varepsilon \, boss \wedge$
 $\neg(y_C \, \varepsilon \, boss) \wedge$
 $\neg(y_V \cong \text{“Secretary”}) \wedge$
 $\neg(y_V \cong \text{“Manager”})).$

/* Employees are components. */
$\forall x_C \, (x_C \, \varepsilon \, components \leftarrow x_C \, \varepsilon \, employees).$
END;

VIEW ***Next_Slots***
DEFINITIONS
 PREDICATE ***before_slots***
 high : {***slots***};
 less : {***slots***};
 END;
UPWARD MAPPING

/* A slot comes before another one if it is more centrally located. */
$\forall x_S, \forall x_V, \forall y_S, \forall y_V$
$(before\_slots(high : x_S, less : y_S) \leftarrow$
$x_S[location : x_V]\ x_S\ \varepsilon\ place \wedge$
$y_S[location : y_V]\ x_S\ \varepsilon\ place \wedge$
$x_V < y_V).$

/* Places are slots. */
$\forall x_S\ (x_S\ \varepsilon\ slots \leftarrow x_S\ \varepsilon\ place).$
END;

3.2.5.3.4 Elementary Inference Actions

/* Check checks whether a state is a possible solution */
ELEMENTARY INFERENCE ACTION ***Check***
PREMISES
States;
CONCLUSIONS
PossibleSolutions;
DEFINITIONS
PREDICATE ***assigned_components***
c : {***components***};
st : {***states***};
END;
PREDICATE ***goal***
g : {***states***};
END;
PREDICATE ***not_assigned_components***
c : {***components***};
st : {***states***};
END;
RULES
/* Every state which fits the goal predicate is a possible solution */
$\forall x_{St}\ (x_{St}\ \varepsilon\ possible\_solutions \leftarrow$
$goal(g : x_{St}) \wedge x_{St}\ \varepsilon\ states).$

/* The predicate not_assigned_components contains a pair (component, state) for each component and state in which a component is not placed by the state. */
$\forall x_C, \forall x_{St}$
$(not\_assigned\_components(c : x_C, s : x_{St}) \leftarrow$
$x_C\ \varepsilon\ components \wedge$
$x_{St}\ \varepsilon\ states \wedge$
$\neg\ assigned\_components(c : x_C, s : x_{St})\ ).$

$\forall x_A, \forall x_C, \forall x_S, \forall x_{St}$
$\quad$ ***(assigned_components(c : x_C, s : x_{St})*** $\leftarrow$
$\quad\quad$ ***x_{St}[assign :: {x_A[c : x_C, s : x_S]}]*** ε ***states).***

/* The predicate goal is true for all states which place all components. */
$\forall x_C, \forall x_{St}$ ***(goal(g : x_{St})*** $\leftarrow$
$\quad\quad$ x_C ε ***components*** $\wedge$
$\quad\quad$ x_{St} ε ***states*** $\wedge$
$\quad\quad$ $\neg$ ***not_assigned_components(c : x_C, s : x_{St})).***

END;

/* Create creates all successor states. */
ELEMENTARY INFERENCE ACTION ***Create***
PREMISES
States, Next_Comp, Next_Slots;
CONCLUSIONS
Successors;
DEFINITIONS
CLASS ***next_comp***
ISA ***possible_comp;***
END;
CLASS ***next_slots***
ISA ***possible_slots;***
END;
CLASS ***possible_comp***
ISA***components***
ELEMENT_ATT
states :: {***states***};
END
CLASS ***possible_slots***
ISA ***slots;***
ELEMENT_ATT
states :: {***states***};
END;
PREDICATE ***assigned_components***
c : {***components***};
st : {***states***};
END;
PREDICATE ***assigned_slots***
s : {***slots***};
st : {***states***};
END;
PREDICATE ***successor_relation***
old : {***states***};
new : {***successors***};

END;
RULES
/* An element of the class successor is a state which is a successor of a current state. */
$\forall x_{St}, \forall y_{St}$
$(y_{St} \;\varepsilon\; \textbf{\textit{successors}} \leftarrow$
$x_{St} \;\varepsilon\; \textbf{\textit{states}} \wedge \textbf{\textit{successor\_relation(old}} : x_{St}, \textbf{\textit{new}} : y_{St})).$

/* A component is possibly the next component to be placed
by the successor states if it has not already been placed by the current state. */
$\forall x_C, \forall x_{St}$
$(x_C[\textbf{\textit{states}} :: \{x_{St}\}] \;\varepsilon\; \textbf{\textit{possible\_comp}} \leftarrow$
$x_{St} \;\varepsilon\; \textbf{\textit{states}} \wedge$
$x_C \;\varepsilon\; \textbf{\textit{components}} \wedge$
$\neg \textbf{\textit{assigned\_components(c}} : x_C, \textbf{\textit{st}} : x_{St})).$
$\forall x_A, \forall x_C, \forall x_{St}$
$(\textbf{\textit{assigned\_components(c}} : x_C, \textbf{\textit{st}} : x_{St}) \leftarrow$
$x_{St}[\textbf{\textit{assign}} :: \{x_A[c : x_C]\}] \;\varepsilon\; \textbf{\textit{states}}).$

/* Select the components to be considered next of all possible ones. */
$\forall x_C, \forall x_{St}, \forall y_C$
$(x_C \;\varepsilon\; \textbf{\textit{next\_comp}}[\textbf{\textit{states}} :: \{x_{St}\}] \leftarrow$
$x_C[\textbf{\textit{states}} :: \{x_{St}\}] \;\varepsilon\; \textbf{\textit{possible\_comp}} \wedge$
$y_C[\textbf{\textit{states}} :: \{x_{St}\}] \;\varepsilon\; \textbf{\textit{possible\_comp}} \wedge$
$\neg \textbf{\textit{before\_comp(high}} : y_C, \textbf{\textit{less}}: x_C)).$

/* A slot is possibly the next slot to be placed by the successors of
a state if it has not already been assigned in the current state. */
$\forall x_S, \forall x_{St}$
$(y_S [\textbf{\textit{states}} :: \{x_{St}\}] \;\varepsilon\; \textbf{\textit{possible\_slots}} \leftarrow$
$x_{St} \;\varepsilon\; \textbf{\textit{states}} \wedge$
$x_S \;\varepsilon\; \textbf{\textit{slots}} \wedge$
$\neg \textbf{\textit{assigned\_slots(s}} : x_S, \textbf{\textit{st}} : x_{St})).$
$\forall x_A, \forall x_S, \forall x_{St}$
$(\textbf{\textit{assigned\_slots(s}} : x_S, \textbf{\textit{st}} : x_{St}) \leftarrow$
$x_{St}[\textbf{\textit{assign}} :: \{x_A[s : x_S] \;\varepsilon\; \textbf{\textit{states}}).$

/* Select the slots to be considered next of all possible slots for a state. */
$\forall x_S, \forall x_{St}, \forall y_S$
$(x_S[\textbf{\textit{states}} :: \{x_{St}\}] \;\varepsilon\; \textbf{\textit{next\_slots}} \leftarrow$
$x_S[\textbf{\textit{states}} : x_{St}] \;\varepsilon\; \textbf{\textit{possible\_slots}} \wedge$
$y_S [\textbf{\textit{states}} : x_{St}] \;\varepsilon\; \textbf{\textit{possible\_slots}} \wedge$
$\neg \textbf{\textit{before\_slots(high}} : y_S, \textbf{\textit{less}}: x_S)).$

/* The successor_relation contains a tuple with an old state and a new state for every possible assignment of the selected components and slots. In addition, also the new state contains this assignment. */

$\forall x_A, \forall x_C, \forall x_S, \forall x_{St}, \forall y_S$
  (successor_relation(old : x_{St}*, new : NewSt(*x_{St}, x_S, x_C*)*
  [assign :: {NewA(x_C, x_S*)[c :* x_C*, s :* x_S*]}]) ←*
    x_C[*states :* x_{St}] ε ***next_comp*** ∧
    x_S[*states :* x_{St}] ε ***next_slots*** ∧
    x_{St} ε ***states).***

END;

/* Init creates the initial states. */
ELEMENTARY INFERENCE ACTION ***Init***
  PREMISES
    States, Next_Comp, Next_Slots;
  CONCLUSIONS
    States;
  DEFINITIONS
    CLASS ***next_comp***
      ISA***possible_comp;***
    END;
    CLASS ***next_slots***
      ISA***possible_slots;***
    END;
  RULES
    /* The initial states are derived. */
    $\forall x_C, \forall x_S, \forall x_{St}$
      (NewSt(nil,x_S, x_C*))[assign :: {NewA(*x_C, x_S*)[c :* x_C*, s :* x_S*]}]*
      ε ***states*** ←
        x_C ε ***next_comp*** ∧
        x_S ε ***next_slots***).
    $\forall x_S, \forall y_S$ (x_S ε ***next_slots*** ←
        x_S ε ***slots*** ∧ y_S ε ***slots*** ∧ ¬***before_slots(high :*** y_S***, less:*** x_S)).
    $\forall x_C \forall y_C$(x_C ε ***next_comp*** ←
        x_C ε ***components*** ∧ y_C ε ***components*** ∧
        ¬***before_comp(high :*** y_C***, less:*** x_C)).

    /* The following formula ensures well-typing. */
    $\forall x_C, \forall x_S$ *(NewA(*x_C, x_S*)* ε ***assignments*** ← x_C ε ***components*** ∧ x_S ε ***slots)*** .

END;

/* Prune checks successor states whether they are correct. If they are, they are regarded as new states. */
ELEMENTARY INFERENCE ACTION ***Prune***
PREMISES
Correct_States, Successors;
CONCLUSIONS
States;
RULES
/* Every successor state which is correct is adopted as new element of state */
$\forall \boldsymbol{x_{St}}\ (\boldsymbol{x_{St}} \in \textbf{\textit{states}} \leftarrow \boldsymbol{x_{St}} \in \textbf{\textit{successors}} \wedge \boldsymbol{x_{St}} \in \textbf{\textit{correct\_state}}).$
END;

ELEMENTARY INFERENCE ACTION ***Select_Solution***
PREMISES
PossibleSolutions;
CONCLUSIONS
Solution;
DEFINITIONS
CLASS ***solution***
ISA ***possible_solutions*** ;
END;
RULES
/* One element of all possible solutions is selected. */
$\forall \boldsymbol{x_{St}}, \forall \boldsymbol{y_{St}}\ (\boldsymbol{x_{St}} \in \textbf{\textit{solution}} \leftarrow$
$\boldsymbol{x_{St}} \in \textbf{\textit{possible\_solutions}} \wedge$
$\boldsymbol{y_{St}} \in \textbf{\textit{possible\_solutions}} \wedge$
$\neg (\boldsymbol{y_{St}} < \boldsymbol{x_{St}})).$
END;

3.3 The Task Layer

Experiences in XCON (cf. [SBJ87], [BaS89]) have shown that great problems arise if the control flow is implicitly specified. Therefore the control flow has been made explicit in the new implementation of XCON in RIME. In KADS (cf. [WSB92]), knowledge about the control flow is also clearly separated from other kinds of knowledge by describing it at a separate layer of the model of expertise, that is at the task layer.

P-KARL is used at the task layer in order to describe which inference actions specified at the inference layer have to be drawn, and when they have to be drawn, i.e. it specifies sequence, iteration, and alternative of inference actions. The following introduces the language primitives, their semantics, and the task layer of the Sisyphus example.

3.3.1 Language Primitives at the Task Layer

The task layer is described by an ***alphabet of a task layer***, which is an alphabet of a P-KARL language with a predefined interpretation.

Def. 115 Alphabet of a task layer

An ***alphabet of a task layer*** is a P-KARL alphabet P = (F, X, B, PS) where

- F = EIA, every function name is the name of an elementary inference action of the inference layer,
- X = $^{I}S \cup \{o\}$, every variable name is the name of a store of the inference layer or refers to the output-data part of the domain layer,
- PS is a finite set of unary predicate symbols $\{\varnothing(\_,c) \mid c \in {}^{D}\Sigma \cup {}^{I}\Sigma$ with ${}^{s}D \models c[[]]$ for a $s \in {}^{I}S \cup \{o\}\}$[1] $\cup \{True(\_)\}$

(cf. Def. 99 “Alphabet of an inference layer” at page 148)).

The function symbols f_i are given by the names of the elementary inference actions, and the variables s_j are given by the store names of the inference layer. The set of boolean variables B has to be defined at the task layer. The predicate symbols will be explained further on.

Primitive programs, which are defined next, are the elementary components at the task layer.

Def. 116 Primitive program

- A primitive program
  $(y_1, ..., y_{j_i}) := f_i\,(x_1, ..., x_{k_i})$
  corresponds to an elementary inference action f_i defined at the inference layer and is

1. The class definitions of the store “s” of the output-data part “o” must contain a class definition for “c” (cf. Def. 101 “Store” at page 153).

called the ***call of an elementary inference action***. The x_l, $l = 1,..., k_i$ must denote all its input stores, and the y_v, $v = 1,..., j_i$ must denote all its output stores.

- A primitive program can be used to define the value of a boolean variable:
  b := FALSE or b := TRUE.
- These are all primitive programs.

In the following, some additional syntactical sugar is introduced which improves the readability of a task layer.

Remark 7 Extended syntax for calls of elementary inference actions

• Instead of

$$(y_1, ..., y_{j_i}) := f_i\,(x_1, ..., x_{k_i})$$

it is

$$(\text{STORES}: y_1, ..., y_{j_i}) := f_i\,(\text{STORES}: x_1, ..., x_{k_i})$$

• If an elementary inference action f_i has the views $v_1,..., v_{v_i}$ as premises, it is:

$$(\text{STORES}: y_1, ..., y_{j_i}) := f_i\,(\text{STORES}: x_1, ..., x_{k_i}; \text{VIEWS}: v_1, ..., v_{v_i})$$

• If an elementary inference action f_i^A has the terminators $t_1,..., t_{t_i}$ as conclusions, it is:

$$(\text{STORES}: y_1,...,y_{j_i}; \text{TERMINATORS}: t_1, ...,t_{t_i}) := f_i\,(\text{STORES}: x_1, ..., x_{k_i})$$

• A function corresponding to an elementary inference action which has one or more terminators would need the variable “o” as a variable, as it assigns a value by changing the contents of “o”. In addition, the newly computed assignment of “o” depends on the old assignment (i.e., it is the union of the old contents and newly derived facts), therefore the function has the variable “o” as an input and output variable:

$$(..., o,...) := f_i(..., o,...)$$

As this item is redundant to the information that the inference action has a terminator as variable this detail is neglected.

So far, the elementary components at the task layer have been defined, which are assignments of elementary inference actions or assignments of truth values to boolean variables. Now, a control flow between these components is defined. Formulae are used to define the control of loops and alternatives.

Def. 117 Formulae

- ***Elementary formulae*** are open formulae $\{\varnothing(s,c) \mid s \in {}^{I}S \cup \{o\}$ and $c \in {}^{D}\Sigma \cup {}^{I}\Sigma$ with ${}^{s}D \models c[[]]\} \cup \{True(b) \mid b \in B\}$.
- ***Composed formulae*** can be built up of elementary formulae by the logical connectives “¬, ∨, ∧“.

A formula $\varnothing$(s,c) can be used to ask whether a store or the output-data part contains an element of the class “c”, i.e. an is-element-of term with “c”. Therefore, “c” must be defined for this store or for the output-data part.

Def. 118 Control flow

Primitive programs “p”, “q” and formulae ψ can be arranged to

- ***Sequence***: p;q
- ***Loop***: WHILE ψ DO p ENDDO
- ***Loop***: REPEAT p UNTIL ψ
- ***Alternative***: IF ψ THEN p ELSE q ENDIF

to express the ***control flow*** between them.

Programs may be combined to named ***subtasks***, similar to procedures in programming languages. They correspond to composed inference actions at the inference layer. Therefore, only those calls of elementary inference actions are allowed within a subtask which are an element of the refinement of the corresponding composed inference action. The name of the subtask is given by the corresponding inference action. A subtask can contain other subtasks if the refinement of a composed inference action contains other composed inference actions.

Def. 119 Subtasks

- A ***subtask*** is a subroutine (cf. Def. 68 “Additional syntactical sugar for P-KARL” at page 107) which defines a name for a program “p”. The syntactical structure is represented like the following:
  SUBTASK name
  p
  ENDSUBTASK
  If a program “q” uses “p” as a component it can use the name of the subroutine instead of “p”. The semantics of a subroutine is defined by the semantics of “p”.
- In addition, it must hold (cf. Def. 112 “Composed inference action” at page 162):
  - name $\in$ CIA, i.e. the name of the subtask must be a name of a composed inference action of the inference layer.
  - If f_i is a function name in an assignment in “p” then $f_i \in {}^{name}$InferenceActions.
  - If name´ is a name of a subtask in “p” then name´ $\in {}^{name}$InferenceActions.
- In addition, the notation of a subtask is extended by the names of the views, input stores, output stores, and terminators of the corresponding composed inference action:
  SUBTASK $name_i$
  VIEWS $v_1,...,v_{v_i}$;

INPUT_STORES $x_1,...,x_{j_i}$;
OUTPUT_STORES $y_1,...,y_{k_i}$;
TERMINATORS $t_1,...,t_{t_i}$;
BODY
p
ENDSUBTASK

A program "q" which uses the subtask "$name_i$" as a component has to use the following notation:

$name_i$ (VIEWS:$v_1,...,v_{v_i}$; INPUT_STORES:$x_1,...,x_{j_i}$;
OUTPUT_STORES:$y_1,...,y_{k_i}$; TERMINATORS:$t_1,...,t_{t_i}$)

A ***task*** at the task layer is represented by an ***initialized program*** and is used to denote the entire problem-solving process of a system. The initialization defines a definite starting point for a program.

Def. 120 Task

A task is an initialized program:
init;
name()
where "name" is the name of the top-level inference action of the inference layer.

3.3.2 Semantics

For a P-KARL language P a semantical structure A = ($\wp$(D), F^A, PS^A) is defined by the inference layer and domain layer (cf. section 2.2). In the following D, F^A, and PS^A are defined.

3.3.2.1 The carrier of a task layer

D is defined by the Herbrand base HB of the L-KARL language $^{D \cup I}L$, i.e. the union of the alphabet of the domain layer ^{D}L and the alphabet of the inference layer ^{I}L.

$\wp(D) := \wp(HB)$

$\wp$ is called the ***carrier*** of the task layer.

3.3.2.2 The Dataflow Variables

With every dataflow variable $x \in X$ ($X = {}^{I}S \cup \{o\}$) the set of class definitions ${}^{x}D$ is associated. The current ***contents*** of a store or the output-data part $|x|$ is determined by the value assignment $\nu(x)$.

3.3.2.3 The Functions

Every function symbol $f_i \in F$ is interpreted by a partial function $f_i^A \in F^A$ with

$f_i^A\colon \wp(HB)^{k_i} \rightarrow \wp(HB)^{j_i}$

The elementary inference actions $f_i \in$ EIA and its semantics is used here to define these functions.

Def. 121 Meaning of a call of an elementary inference action

$$\left(y_1 := {}^{1}f_i(x_1, ..., x_{k_i}), ..., y_{j_i} := {}^{j_i}f_i(x_1, ..., x_{k_i})\right)^A = \{(\nu, \nu[f_i]) \mid \nu \in S^A\}$$

$\nu[f_i]$ is a term assignment with:

$$\nu[f_i](x) = \begin{cases} {}^{t}f_i^A(\nu(x_1), ..., \nu(x_{k_i})) & \text{if } x = y_t \text{ for } t = 1, ..., j_i \\ \nu(x) & \text{else.} \end{cases}$$

${}^{t}f_i^A(\nu(x_1), ..., \nu(x_{k_i}))$ is defined as follows:

If the correct well-typed Herbrand model H of f_i^A according to the current contents $|x_1|, ..., |x_{k_i}|$ of its input stores exists (cf. Def. 109 “Correct well-typed perfect Herbrand model of an elementary inference action” at page 159), then the new assignment of $\nu[f_i](y_t)$, $t = 1, ..., j_i$, is the updated content $|y_t|$ of the according output store y_t as defined by Def. 110 “Output of an elementary inference action” at page 160, i.e. $|y_t|$ is the set of all terms of H over the class definitions of y_t.

(Cf. Def. 76 “Interpretation function” at page 117)

3.3.2.4 The Relations

It remains to define the relations which are used to interpret the predicate symbols.

Def. 122 Satisfaction of an elementary formula

- If ψ is an elementary formula $\varnothing(s,c)$, then
  $\psi^A = \{\nu \mid \nu(s)$, i.e. $|s|$, does not contain an is-element-of term $t \,\varepsilon\, c$, $\nu \in S^A\}$.
- If ψ is an elementary formula True(b), then
  $\psi^A = \{\nu \mid \nu(b) = \wp(HB), \nu \in S^A\}$.

(cf. Def. 72 “Satisfaction of a formula” at page 110 for the satisfaction of composed formulae)

3.3.2.5 Subtasks

The semantics of a subtask, i.e. a composed program, is defined by mutual induction over its structure. Therefore, it is defined by the elementary inference actions it contains and the control flow which is defined between them.

3.3.3 Task Layer of the Sisyphus Example

In the following the control flow is defined between the elementary inference actions for a solution of the Sisyphus task.

3.3.3.1 Alphabet

F = {***Check, Create, Init, Prune, Select_Solution***}

X = {***o, PossibleSolutions, States, Successors***}

B = { }

Prop = {∅(PossibleSolutions, possible_solutions), ∅(o, employee), ∅(States, states)}

Additional syntactical sugar:

- Terminators = {***Solution***}
- Views = {***Correct_States, Next_Comp, Next_Slots)***}
- Subtasks = {***Assign***}

3.3.3.2 The Task “Assign”

init; Assign()

3.3.3.3 Refinement of “Assign”

First, the initial states are derived. Then, a loop of several elementary inference actions is executed as long as no solution is found and further states which can be regarded next as a solution exist. In particular, the loop consists of the following activities: First it is checked, whether the current states contain a solution. If yes, a solution is selected and the problem-solving process stops. If no, the successors states are derived and pruned. Then the loop is repeated for these successor states.

SUBTASK ***Assign***
BODY
/*Init computes the initial successors. */
(STORES: States) := Init(VIEWS: Next_Comp,Next_Slots);
REPEAT
/* Check for a state that could be used as a possible solution. */
(STORES: Possible_Solutions) := Check(STORES: States);
IF ∅***(PossibleSolutions, possible_solutions)***
THEN
/* Check did not find a possible solution. Therefore, new states are computed which could be a possible solution. */
(STORES: Successors) := Create(STORES: States; VIEWS: Next_Comp,Next_Slots);
(STORES: States) := Prune(STORES: Successors; VIEWS: Correct_States)
ELSE
/* A solution is selected from the set of possible solutions. */
(TERMINATORS: Solution) := Select_Solution(STORES: Possible_Solutions)
ENDIF
UNTIL (¬∅***(o, employee)*** ∨ ∅***(States, states))***
ENDSUBTASK

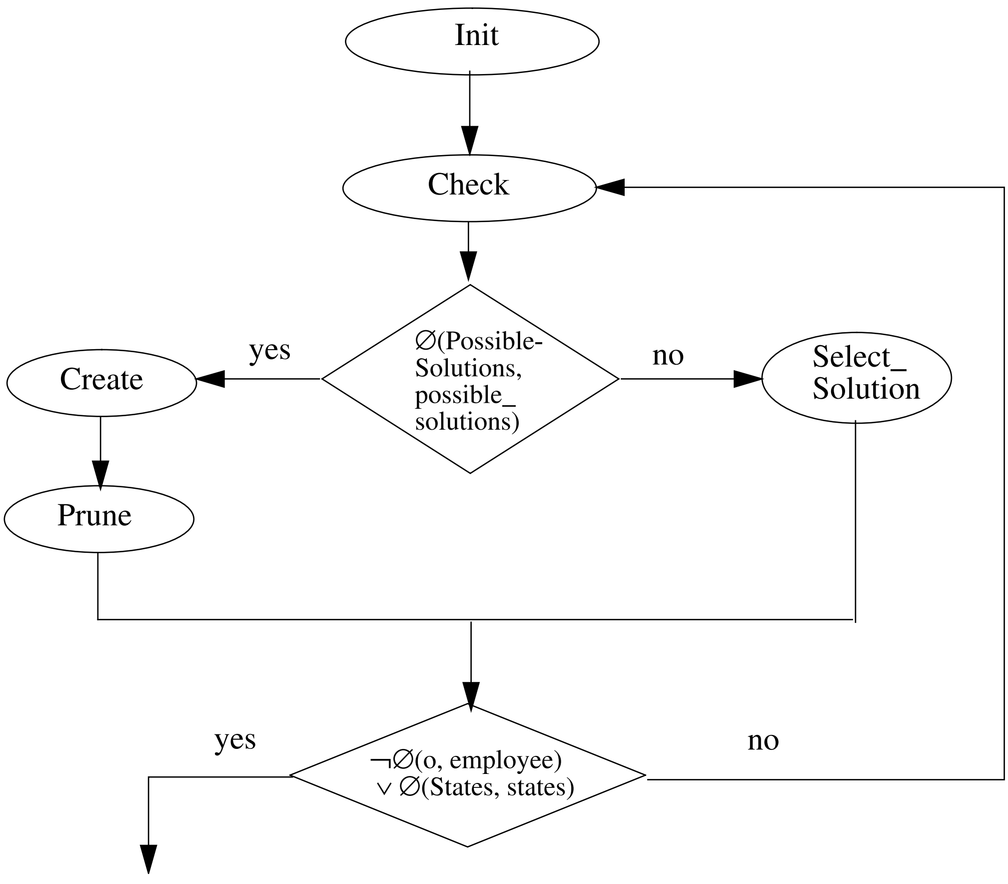


Figure 19. Refinement of the task “Assign”

3.4 A Trace of the Solution of the Sisyphus Example

In the following two manual traces[1] of the solution of the Sisyphus example are shown. The problem-solving method used in this thesis does not guarantee to find a solution even if one exists; this possible case will be dealt with in the first trace, which stops without finding a solution. For the second trace, the domain layer was slightly modified, and a solution was found.

3.4.1 Trace Without a Solution

3.4.1.1 Input-Data Part |i|:

class: employee

E-ID	hacker	name	project	role	smoker	work_ with
wl	True	“Werner”	“RESPECT”	“Researcher”	False	
jl	True	“Jürgen”	“EULISP”	“Researcher”	False	
td	False	“Thomas”	“EULISP”	“Researcher”	False	

class: boss ≤ {employee}

E-ID	hacker	name	project	role	smoker	work_ with

class: big_boss ≤ {boss, employee}

E-ID	hacker	name	project	role	smoker	work_ with
td	False	“Thomas”	“EULISP”	“Researcher”	False	

class: place

E-ID	location	room
p0	0	“c5-113”
p1	0	“c5-113”

1. The interpreter of the new KARL version was not finished at this stage.

class: place

E-ID	location	room
p2	1	“c5-114”
p3	1	“c5-115”

3.4.1.2 Contents of the Stores

Trace of the problem-solving process

Elementary Inference Action	Possible Solutions	States	Successors
Init	∅	jl ε components, td ε components, wl ε components, p0 ε slots, p1 ε slots, p2 ε slots, p3 ε slots, NewA(jl,p0) ε assignments, NewA(jl,p1) ε assignments, NewA(jl,p2) ε assignments, NewA(jl,p3) ε assignments, NewA(td,p0) ε assignments, NewA(td,p1) ε assignments, NewA(td,p2) ε assignments, NewA(td,p3) ε assignments, NewA(wl,p0) ε assignments, NewA(wl,p1) ε assignments, NewA(wl,p2) ε assignments, NewA(wl,p3) ε assignments, NewSt(nil,td,p0)[assign :: {NewA(td,p0)[c : td, s : p0]}] ε states, NewSt(nil,td,p1)[assign :: {NewA(td,p1)[c : td, s : p1]}] ε states	∅
Check	∅	jl ε components,... NewA(wl,p3) ε assignments, NewSt(nil,td,p0)[assign :: {NewA(td,p0)[c : td, s : p0]}] ε states, NewSt(nil,td,p1)[assign :: {NewA(td,p1)[c : td, s : p1]}] ε states	∅

Trace of the problem-solving process

Elementary Inference Action	Possible Solutions	States	Successors
Create	∅	jl ε components,... NewA(wl,p3) ε assignments, NewSt(nil,td,p0)[assign :: {NewA(td,p0)[c : td, s : p0]}] ε states, NewSt(nil,td,p1)[assign :: {NewA(td,p1)[c : td, s : p1]}] ε states	jl ε components,... NewA(wl,p3) ε assignments, NewSt(NewSt(nil,td,p0),jl,p1)[assign :: {NewA(td,p0)[c : td, s : p0], NewA(jl,p1)[c : jl, s : p1]}] ε successors, NewSt(NewSt(nil,td,p0),wl,p1)[assign :: {NewA(td,p0)[c : td, s : p0], NewA(wl,p1)[c : wl, s : p1]}] ε succes- sors, NewSt(NewSt(nil,td,p1),jl,p0)[assign :: {NewA(td,p1)[c : td, s : p1], NewA(jl,p0)[c : jl, s : p0]}] ε successors, NewSt(NewSt(nil,td,p1),wl,p0)[assign :: {NewA(td,p1)[c : td, s : p1], NewA(wl,p0)[c : wl, s : p0]}] ε succes- sors,
Prune	∅	jl ε components,... NewA(wl,p3) ε assignments,	jl ε components,... NewA(wl,p3) ε assignments, NewSt(NewSt(nil,td,p0),jl,p1)[assign :: {NewA(td,p0)[c : td, s : p0], NewA(jl,p1)[c : jl, s : p1]}] ε successors, NewSt(NewSt(nil,td,p0),wl,p1)[assign :: {NewA(td,p0)[c : td, s : p0], NewA(wl,p1)[c : wl, s : p1]}] ε succes- sors, NewSt(NewSt(nil,td,p1),jl,p0)[assign :: {NewA(td,p1)[c : td, s : p1], NewA(jl,p0)[c : jl, s : p0]}] ε successors, NewSt(NewSt(nil,td,p1),wl,p0)[assign :: {NewA(td,p1)[c : td, s : p1], NewA(wl,p0)[c : wl, s : p0]}] ε succes- sors,

The process stops, because ∅(States, states) is evaluated to true as the store “States” does not contain an is-element-of term for the class “states”.

3.4.1.3 Output-Data Part |o|:

class: employee

E-Id	placement

3.4.2 Trace With a Solution

3.4.2.1 Input-Data Part |i|:

class: employee

E-ID	hacker	name	project	role	smoker	work_ with
wl	True	“Werner”	“RESPECT”	“Researcher”	False	
jl	True	“Jürgen”	“EULISP”	“Researcher”	False	
td	False	“Thomas”	“EULISP”	“Researcher”	False	

class: boss ≤ {employee}

E-ID	hacker	name	project	role	smoker	work_ with

class: big_boss ≤ {boss, employee}

E-ID	hacker	name	project	role	smoker	work_ with
td	False	“Thomas”	“EULISP”	“Researcher”	False	

class: place

E-ID	location	room
p0	0	“c5-113”
p1	**1**	**“c5-114”**
p2	**2**	**“c5-115”**
p3	**2**	“c5-115”

3.4.2.2 Contents of the Stores

Trace of the problem-solving process

Elementary Inference Action	Possible Solutions	States	Successors
Init	∅	jl ε components, td ε components, wl ε components, p0 ε slots, p1 ε slots, p2 ε slots, p3 ε slots, NewA(jl,p0) ε assignments, NewA(jl,p1) ε assignments, NewA(jl,p2) ε assignments, NewA(jl,p3) ε assignments, NewA(td,p0) ε assignments, NewA(td,p1) ε assignments, NewA(td,p2) ε assignments, NewA(td,p3) ε assignments, NewA(wl,p0) ε assignments, NewA(wl,p1) ε assignments, NewA(wl,p2) ε assignments, NewA(wl,p3) ε assignments, NewSt(nil,td,p0)[assign :: {NewA(td,p0)[c : td, s : p0]}] ε states	∅
Check	∅	jl ε components,... NewA(wl,p3) ε assignments, NewSt(nil,td,p0)[assign :: {NewA(td,p0)[c : td, s : p0]}] ε states	∅
Create	∅	jl ε components,... NewA(wl,p3) ε assignments, NewSt(nil,td,p0)[assign :: {NewA(td,p0)[c : td, s : p0]}] ε states	jl ε components,... NewA(wl,p3) ε assignments, NewSt(NewSt(nil,td,p0),jl,p1) [assign :: {NewA(td,p0)[c : td, s : p0], NewA(jl,p1)[c : jl, s : p1]}] ε successors, NewSt(NewSt(nil,td,p0),wl, p1)[assign :: {NewA(td,p0)[c : td, s : p0], NewA(wl,p1)[c : wl, s : p1]}] ε successors
Prune	∅	jl ε components,... NewA(wl,p3) ε assignments, NewSt(NewSt(nil,td,p0),jl,p1) [assign :: {NewA(td,p0)[c : td, s : p0], NewA(jl,p1)[c : jl, s : p1]}] ε states, NewSt(NewSt(nil,td,p0),wl, p1)[assign :: {NewA(td,p0)[c : td, s : p0], NewA(wl,p1)[c : wl, s : p1]}] ε states	jl ε components,... NewA(wl,p3) ε assignments, NewSt(NewSt(nil,td,p0),jl,p1) [assign :: {NewA(td,p0)[c : td, s : p0], NewA(jl,p1)[c : jl, s : p1]}] ε successors, NewSt(NewSt(nil,td,p0),wl, p1)[assign :: {NewA(td,p0)[c : td, s : p0], NewA(wl,p1)[c : wl, s : p1]}] ε successors

Trace of the problem-solving process

Elementary Inference Action	Possible Solutions	States	Successors
Check	∅	jl ε components,... NewA(wl,p3) ε assignments, NewSt(NewSt(nil,td,p0),jl,p1) [assign :: {NewA(td,p0)[c : td, s : p0], NewA(jl,p1)[c : jl, s : p1]}] ε states, NewSt(NewSt(nil,td,p0),wl, p1)[assign :: {NewA(td,p0)[c : td, s : p0], NewA(wl,p1)[c : wl, s : p1]}] ε states	jl ε components,... NewA(wl,p3) ε assignments, NewSt(NewSt(nil,td,p0),jl,p1) [assign :: {NewA(td,p0)[c : td, s : p0], NewA(jl,p1)[c : jl, s : p1]}] ε successors, NewSt(NewSt(nil,td,p0),wl, p1)[assign :: {NewA(td,p0)[c : td, s : p0], NewA(wl,p1)[c : wl, s : p1]}] ε successors
Create	∅	jl ε components,... NewA(wl,p3) ε assignments, NewSt(NewSt(nil,td,p0),jl,p1) [assign :: {NewA(td,p0)[c : td, s : p0], NewA(jl,p1)[c : jl, s : p1]}] ε states, NewSt(NewSt(nil,td,p0),wl, p1)[assign :: {NewA(td,p0)[c : td, s : p0], NewA(wl,p1)[c : wl, s : p1]}] ε states	jl ε components,... NewA(wl,p3) ε assignments, NewSt(NewSt(NewSt(nil,td,p0),jl,p1),wl,p2) [assign :: {NewA(td,p0)[c : td, s : p0], NewA(jl,p1)[c : jl, s : p1], NewA(wl,p2)[c : wl, s : p2]}] ε successors, NewSt(NewSt(NewSt(nil,td,p0),jl,p1),wl,p3) [assign :: {NewA(td,p0)[c : td, s : p0], NewA(jl,p1)[c : jl, s : p1], NewA(wl,p3)[c : wl, s : p3]}] ε successors, NewSt(NewSt(NewSt(nil,td,p0),wl,p1),tj,p2)[assign :: {NewA(td,p0)[c : td, s : p0], NewA(wl,p1)[c : wl, s : p1], NewA(jl,p2)[c : jl, s : p2]}] ε successors, NewSt(NewSt(NewSt(nil,td,p0),wl,p1),tj,p3)[assign :: {NewA(td,p0)[c : td, s : p0], NewA(wl,p1)[c : wl, s : p1], NewA(jl,p3)[c : jl, s : p3]}] ε successors,

Trace of the problem-solving process

Elementary Inference Action	Possible Solutions	States	Successors
Prune	∅	jl ε components,... NewA(wl,p3) ε assignments, NewSt(NewSt(NewSt(nil,td,p0),jl,p1),wl,p2) [assign :: {NewA(td,p0)[c : td, s : p0], NewA(jl,p1)[c : jl, s : p1], NewA(wl,p2)[c : wl, s : p2]}] ε states, NewSt(NewSt(NewSt(nil,td,p0),jl,p1),wl,p3) [assign :: {NewA(td,p0)[c : td, s : p0], NewA(jl,p1)[c : jl, s : p1], NewA(wl,p3)[c : wl, s : p3]}] ε states, NewSt(NewSt(NewSt(nil,td,p0),wl,p1),tj,p2)[assign :: {NewA(td,p0)[c : td, s : p0], NewA(wl,p1)[c : wl, s : p1], NewA(jl,p2)[c : jl, s : p2]}] ε states, NewSt(NewSt(NewSt(nil,td,p0),wl,p1),tj,p3)[assign :: {NewA(td,p0)[c : td, s : p0], NewA(wl,p1)[c : wl, s : p1], NewA(jl,p3)[c : jl, s : p3]}] ε states,	jl ε components,... NewA(wl,p3) ε assignments, NewSt(NewSt(NewSt(nil,td,p0),jl,p1),wl,p2) [assign :: {NewA(td,p0)[c : td, s : p0], NewA(jl,p1)[c : jl, s : p1], NewA(wl,p2)[c : wl, s : p2]}] ε successors, NewSt(NewSt(NewSt(nil,td,p0),jl,p1),wl,p3) [assign :: {NewA(td,p0)[c : td, s : p0], NewA(jl,p1)[c : jl, s : p1], NewA(wl,p3)[c : wl, s : p3]}] ε successors, NewSt(NewSt(NewSt(nil,td,p0),wl,p1),tj,p2)[assign :: {NewA(td,p0)[c : td, s : p0], NewA(wl,p1)[c : wl, s : p1], NewA(jl,p2)[c : jl, s : p2]}] ε successors, NewSt(NewSt(NewSt(nil,td,p0),wl,p1),tj,p3)[assign :: {NewA(td,p0)[c : td, s : p0], NewA(wl,p1)[c : wl, s : p1], NewA(jl,p3)[c : jl, s : p3]}] ε successors,

Trace of the problem-solving process

Elementary Inference Action	Possible Solutions	States	Successors
Check	jl ε components,... NewA(wl,p3) ε assignments, NewSt(NewSt(NewSt(nil,td,p0),jl,p1),wl,p2) [assign :: {NewA(td,p0)[c : td, s : p0], NewA(jl,p1)[c : jl, s : p1], NewA(wl,p2)[c : wl, s : p2]}] ε possible_solution, NewSt(NewSt(NewSt(nil,td,p0),jl,p1),wl,p3) [assign :: {NewA(td,p0)[c : td, s : p0], NewA(jl,p1)[c : jl, s : p1], NewA(wl,p3)[c : wl, s : p3]}] ε possible_solution, NewSt(NewSt(NewSt(nil,td,p0),wl,p1),tj,p2)[assign :: {NewA(td,p0)[c : td, s : p0], NewA(wl,p1)[c : wl, s : p1], NewA(jl,p2)[c : jl, s : p2]}] ε possible_solution, NewSt(NewSt(NewSt(nil,td,p0),wl,p1),tj,p3)[assign :: {NewA(td,p0)[c : td, s : p0], NewA(wl,p1)[c : wl, s : p1], NewA(jl,p3)[c : jl, s : p3]}] ε possible_solution,	jl ε components,... NewA(wl,p3) ε assignments, NewSt(NewSt(NewSt(nil,td,p0),jl,p1),wl,p2) [assign :: {NewA(td,p0)[c : td, s : p0], NewA(jl,p1)[c : jl, s : p1], NewA(wl,p2)[c : wl, s : p2]}] ε states, NewSt(NewSt(NewSt(nil,td,p0),jl,p1),wl,p3) [assign :: {NewA(td,p0)[c : td, s : p0], NewA(jl,p1)[c : jl, s : p1], NewA(wl,p3)[c : wl, s : p3]}] ε states, NewSt(NewSt(NewSt(nil,td,p0),wl,p1),tj,p2)[assign :: {NewA(td,p0)[c : td, s : p0], NewA(wl,p1)[c : wl, s : p1], NewA(jl,p2)[c : jl, s : p2]}] ε states, NewSt(NewSt(NewSt(nil,td,p0),wl,p1),tj,p3)[assign :: {NewA(td,p0)[c : td, s : p0], NewA(wl,p1)[c : wl, s : p1], NewA(jl,p3)[c : jl, s : p3]}] ε states,	jl ε components,... NewA(wl,p3) ε assignments, NewSt(NewSt(NewSt(nil,td,p0),jl,p1),wl,p2) [assign :: {NewA(td,p0)[c : td, s : p0], NewA(jl,p1)[c : jl, s : p1], NewA(wl,p2)[c : wl, s : p2]}] ε successors, NewSt(NewSt(NewSt(nil,td,p0),jl,p1),wl,p3) [assign :: {NewA(td,p0)[c : td, s : p0], NewA(jl,p1)[c : jl, s : p1], NewA(wl,p3)[c : wl, s : p3]}] ε successors, NewSt(NewSt(NewSt(nil,td,p0),wl,p1),tj,p2)[assign :: {NewA(td,p0)[c : td, s : p0], NewA(wl,p1)[c : wl, s : p1], NewA(jl,p2)[c : jl, s : p2]}] ε successors, NewSt(NewSt(NewSt(nil,td,p0),wl,p1),tj,p3)[assign :: {NewA(td,p0)[c : td, s : p0], NewA(wl,p1)[c : wl, s : p1], NewA(jl,p3)[c : jl, s : p3]}] ε successors,

The last inference action to be executed is “Select Solution”. It modifies the contents of the output-data part |o|.

3.4.2.3 Output-Data Part |o|:

class: employee

E-ID	placement
jl	p1
td	p0
wl	p2

3.5 A Strategic Layer in KARL or an Alternative View at the Task Layer

An alternative to the representation of control flow by a procedural language are Petri nets (cf. [Gen86], [Rei86]). The application of Petri nets to represent control knowledge has some advantages:

- Petri nets are a formalism to specify causal relationships between activities. They implicitly specify control flow as defined by the causal relationships. Thus, causal relationships and dependencies between tasks and their execution order can be specified. Using a procedural programming language the control flow must be explicitly implemented.
- The well-defined semantics and the corresponding tools of Petri nets can be used to analyse the specified control knowledge.
- Petri nets are a useful and easy-to-read graphical representation formalism.
- Petri nets are a well-applied representation formalism even for complex control flow dependencies of information systems (cf. [LNO89], [Stu87]).
- It is easy to express parallelism with Petri nets. Tasks which are causally independent of each other can be done in parallel.
- Temporal constraints and integrity constraints for the control flow can easily be represented declaratively (cf. [Obe90]).

Therefore, Petri nets are proposed here for the ***strategic layer*** of a model of expertise.

> "Strategic knowledge determines what goals are relevant to solve a particular problem. How each goal is achieved is determined by the task knowledge." [WSB92], p. 25

If a task is identified with the goal it helps to reach one might say:

> ***The strategic layer specifies the causal relationships of several tasks with the aim of solving a given problem. How a single task achieves this goal is not a matter of interest here.***

The problem is now how Petri nets can be related to a strategic layer or to an alternative representation of the task layer:

- A ***task*** or ***inference action*** is represented by a ***transition***.
- A ***store*** is represented by a ***place***.
- A transition is ***connected*** with a place if the corresponding inference action contains the corresponding store as input store (premise) or output store (conclusion).
- ***Marks*** which label a place represent the elements of the corresponding store.
- A task can be ***executed*** if the elements of the input stores fit the logical description of the task. This logical description could be the description of an elementary inference action or additional pre-conditions, especially in the case of a composed inference action. The execution of a task can change the contents of its output stores.

- To achieve a control flow additional transitions and places with no correspondence at the inference layer can be introduced. Places can be used to store control information, for example.
- Petri nets allow ***hierarchical refinement***, which permits the representation at different levels of refinement reflecting the hierarchical structure of the inference layer.

Causal relationships between tasks can be specified by a Petri net. Their temporal dependencies, i.e. their sequence of execution, is only specified as a consequence of their causal ordering. It is not longer necessary to implement this ordering explicitly.

3.6 Model of Cooperation

"The KADS model of expertise can be viewed as an autistic problem solver" [GrB92], p. 95. The cooperation with other agents is modelled by the so-called ***model of cooperation*** (cf. [GrB92]). It is derived from the task model. A task model[1] specifies the whole task and decomposes it into subtasks, which can be mapped on single agents, like the system or the user. The identification of interdependencies among the subtasks indicates when and how an agent solving a task has to cooperate with others. The details of a task model and how the model of cooperation is derived from it will, however, not be dealt with any further (cf. [GrB92]). Here, only aspects of the integration of the model of cooperation into the model of expertise will be discussed. The model which integrates both is called the ***conceptual model***. It models the problem-solving process as it is done by the knowledge-based systems in cooperation with other agents, the user, for example. The model of cooperation is not a tool for specifying the communication with the knowledge-based system, i.e. the specification of a sophisticated user interface. Communication is a means of cooperation, but the cooperation of several agents can be specified without referring to their ways of communication. The underlying assumption is that an adequate way of communication is possible.

[GrB92] distinguish whether the user or the system has the ***initiative*** for the cooperation. In the following only the second case is discussed because no means are introduced which could be used to interrupt a problem-solving process modelled in KARL. Therefore, always the system has the initiative. [GrB92] distinguish further whether the system ***presents*** information or whether it ***obtains*** information. Both cases are possible with KARL. In extension to [GrB92] two kinds of interactions are distinguished here:

- One agent needs ***data*** from a second agent.
- One agents needs ***control information*** from a second agent.

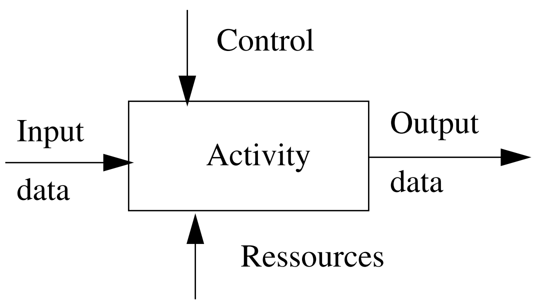


Figure 20. An activity diagram in SADT

The distinction between dataflow and control flow is common in software engineering and is also captured by the difference of inference layer and task layer. In SADT (cf. [Ros77]), for example, data and control flow are distinguished in the following manner (cf. figure 20): The input data are transformed to the output data by an activity; the control information guides this transformation process but is not transformed itself. The control information is carried by a

1. "The creation of a task model involves decomposition of the task, identification of interdependencies among subtasks and the distribution of subtasks over the agents ´system´ and one or more ´user types´."[GrB92], p. 92

small amount of bites, i.e. in KARL one bit is enough to carry the control information. Similar to the distinction of inference layer and task layer, the exchange of control information is represented only at the task layer, whereas the exchange of data is represented at both layers, i.e. at the task layer it is represented when the interactions take place.

3.6.1 Data

The first kind of communication is modelled at the inference layer by additional elementary inference actions. At the task layer it is specified when one of these interactions takes place. Therefore, these elementary inference actions can also be used as elementary components at the task layer.

Two interactions are distinguished, and each of them is modelled as a special elementary inference action:

- ***obtain***: The system consumes input.
- ***present***: The system produces output.

3.6.1.1 The Stores User

For the two elementary inference actions "artificial stores", called ***user***, are defined. ***obtain*** applies a *user* as an input store and ***present*** as an output store. Several of these stores can be defined (cf. section 3.2.2.3).

Def. 123 Store *user*

> A ***store*** $user_i$ is a tuple $({}^{user_i}D)$ where ${}^{user_i}D$ is a set of class definitions over the alphabet ${}^{D \cup I}L$.[1]

The set ${}^{user_i}D$ of class definitions is used to determine the contents of the role $user_i$. Sets of facts are only allowed as contents of a store if they fit these definitions. The syntactical structure of a store $user_i$ is as follows:

```
STORE useri
        DEFINITIONS
                ...;
END
```

1. ${}^{D \cup I}L$ is the union of the alphabet of the domain layer ${}^{D}L$ and of the alphabet of the inference layer ${}^{I}L$.

3.6.1.2 Obtain

The elementary inference action ***obtain*** is used to deliver data to the system. The contents of the output stores of *obtain* are refilled according to the input given by the store *user*.

Def. 124 An elementary inference action *obtain*

An ***elementary inference action obtain*** is a tuple obtain = $(\{{}^{obtain}user\}, {}^{obtain}OutputStores, {}^{obtain}D, {}^{obtain}R, {}^{obtain}C)$ where

- ${}^{obtain}user$ is a name of a store $user_i$ which is the premise of *obtain*,
- ${}^{obtain}OutputStores \subseteq {}^{I}S$ is a set of store names and defines output stores, i.e. conclusions, of *obtain*,
- ${}^{obtain}D$ is a set of class definitions and predicate definitions over the alphabet ${}^{D\cup I}L$,
- ${}^{obtain}R$ is a set of rules over the alphabet ${}^{D\cup I}L$, and
- ${}^{obtain}C$ is a pair of two sets of constraints $({}^{obtain}C_1, {}^{obtain}C_2)$ over the alphabet ${}^{D\cup I}L$.

(cf. Def. 105 “Elementary inference action” at page 156)

Thus, *obtain* is an elementary inference action with no terminators and views and with one input store *user*.

The syntactical structure of *obtain* at the inference layer is as follows:

```
ELEMENTARY INFERENCE ACTION obtain
    PREMISES
        user;
    CONCLUSIONS
        ...;
    DEFINITIONS
        ...;
    RULES
        ...;
    NECESSARY CONDITIONS
        ...;
END
```

The syntactical structure of *obtain* at the task layer is:

(STORES: $store_1$,..., $store_j$) := obtain(STORES: user)

The definition of the ***input***, of a ***correct well-typed perfect Herbrand model***, and of the ***output*** of *obtain* are identical to the definitions in section 3.2.3.1.

Remark 8 Multiple occurrence of *obtain*

Multiple instances of *obtain* in an inference structure can be distinguished by numeration.

3.6.1.3 Present

The elementary inference action ***present*** is used to deliver data to an external agent. The content of the output store *user* of *present* is refilled according to the model of *present*.

Def. 125 An elementary inference action *present*

An ***elementary inference action present*** is a tuple present = (^{present}Views, ^{present}InputStores, {^{present}user}, ^{present}D, ^{present}R, ^{present}C) where

- ^{present}Views $\subseteq$ ^{I}V is a possibly empty set of view names and defines views (i.e., premises) of *present*,
- ^{present}InputStores $\subseteq$ ^{I}S is a possibly empty set of store names and defines input stores (i.e., premises) of *present*,
- ^{present}user is a name of a store $user_i$ which is the conclusion of *present*,
- ^{present}D is a set of class definitions and predicate definitions over the alphabet $^{D\cup I}$L,
- ^{present}R is a set of rules over the alphabet $^{D\cup I}$L, and
- ^{present}C is a pair of two sets of constraints ($^{present}C_1$, $^{present}C_2$) over the alphabet $^{D\cup I}$L.

(cf. Def. 105 “Elementary inference action” at page 156)

Thus, *present* is an elementary inference action with no terminators and one output store user.

The syntactical structure of *present* at the inference layer is as follows:

```
ELEMENTARY INFERENCE ACTION present
    PREMISES
        ...;
    CONCLUSIONS
        user;
    DEFINITIONS
        ...;
    RULES
        ...;
    NECESSARY CONDITIONS
        ...;
END
```

The syntactical structure of *present* at the task layer is:

(STORES: $user_i$) := present(STORES: $store_1$,..., $store_j$; VIEWS:v_1,..., v_v)

The definition of the ***input***, of a ***correct well-typed perfect Herbrand model***, and of the ***output*** of *present* are identical to the definitions in section 3.2.3.2.

Remark 9 Multiple occurrence of *present*

Multiple instances of *present* in an inference structure can be distinguished by numeration.

3.6.2 Control information

As information concerning the control flow is not represented at the inference layer, the following language construct is only available at the task layer.

It may be possible that the systems requires control information from a user. An elementary component
b := ask()
is offered which reads the value TRUE or FALSE from the input and can be used to control the further task execution by storing this value as the new variable assignment of “b”.

As already mentioned, no control flow from the system to the user is modelled here.

Chapter 4

KARL at the Graphical Level

As a rule, formal descriptions do not give a vivid impression of ideas and facts. Therefore, formal specifications are not an adequate tool for communicating with experts or system users. KARL offers graphical representations of most modelling primitives to improve their understandability. Every graphical symbol has a defined meaning given by the semantics of the corresponding language primitive:

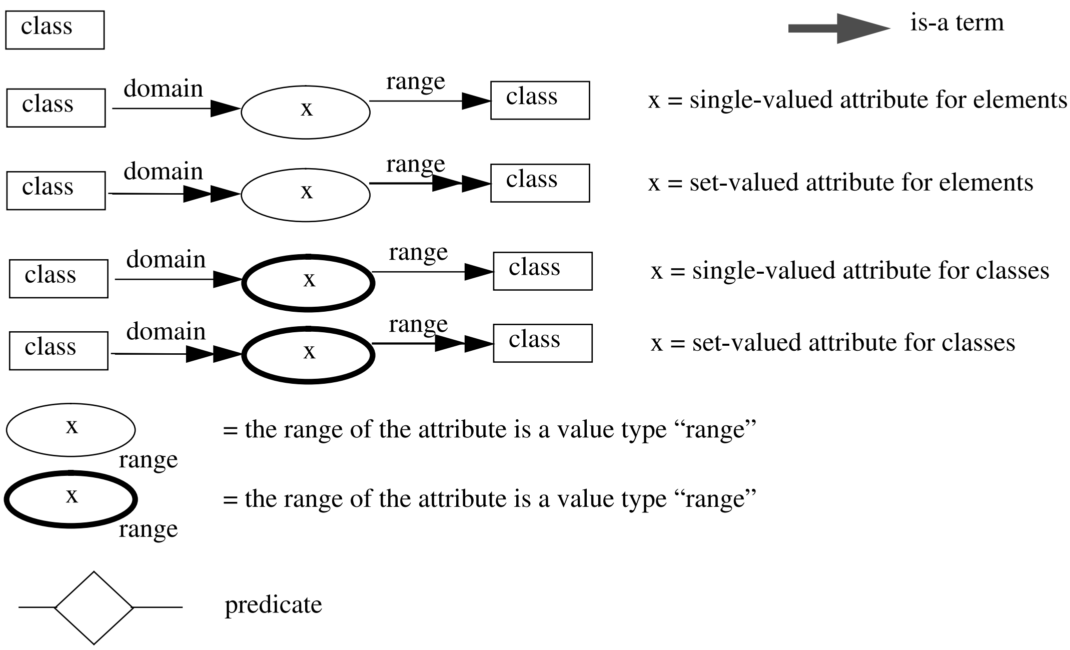
- A variant of Enhanced-Entitity-Relationship (EER) diagrams (cf. [ElN89]) for the domain layer,
- a variant of levelled dataflow diagrams (cf. [You89]) for the inference layer, and
- a variant of programflow diagrams (cf. [Din66001]) for the task layer.

Specifications of real-world systems become unintelligible because of their complexity. Therefore all three graphical representations include hierarchical refinement to allow to represent the system on different levels of refinement.

The chapters 2, 3, and 5 contain several examples using graphical representation of the language primitives.

4.1 The Domain Layer

Enhanced-Entity-Relationship (EER) diagrams (cf. [ElN89], [HuK87], for example) are well-known techniques of graphically representing static knowledge. A similar representation is used here for the domain layer of KARL. Class terms and is-a terms are represented graphically. Additionally, ground data and is-element-of terms are represented by tables. In figure 21 the graphical primitives are shown.



class: $c \leq \{..., c_i, ...\}$

E-ID	attribute$_1$	...	attribute$_k$

Figure 21. Graphical representation at the domain layer.

Rules and constraints are not represented graphically. This would, however, be possible by applying the ideas given in [Cru92] that allow the visualization of F-terms and the introduction of user-defined icons.

4.2 The Inference Layer

The terminological knowledge which is defined in inference actions and roles can be represented by the same graphical primitives as introduced at the domain layer.

As, in the interpretation of KARL, the inference layer is similar to a dataflow diagram (cf. [You89]), a similar kind of formalism is used for it.

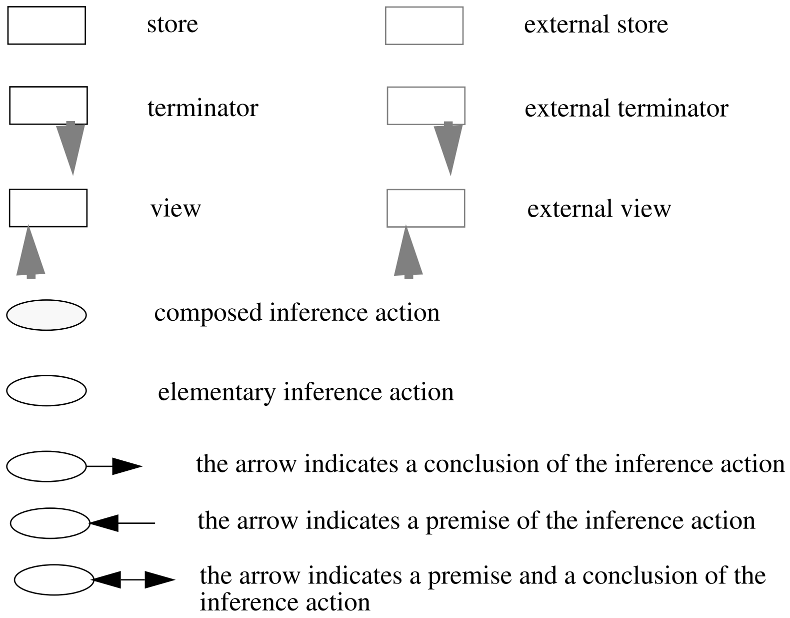
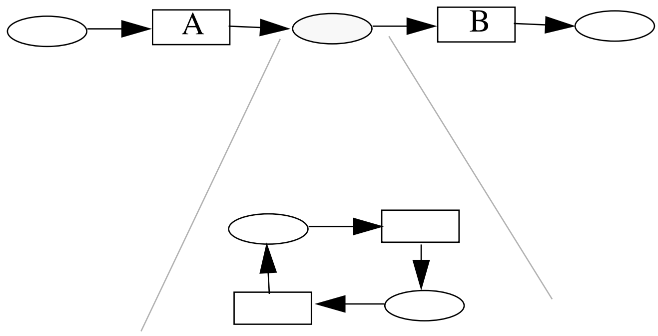


Figure 22. Graphical representation at the inference layer.

Similarly to dataflow diagrams and in extension of the original proposal of the KADS-I group (cf. [WSB92]), the inference layer is extended here by graphical refinement (cf. section 3.2.4.2). The rules that apply for a consistent refinement are comparable to those applying in the domain of dataflow diagrams or Petri nets (cf. Def. 114 “Roles of a well-defined composed inference action” at page 164). In figure 21, some wrong refinements of composed inference actions are shown:

If a cia has a premise (A) or a conclusion (B) then its refinement must have them as external roles.



If no further inference action besides the cia refer to a role (A) then the role must be an internal role.

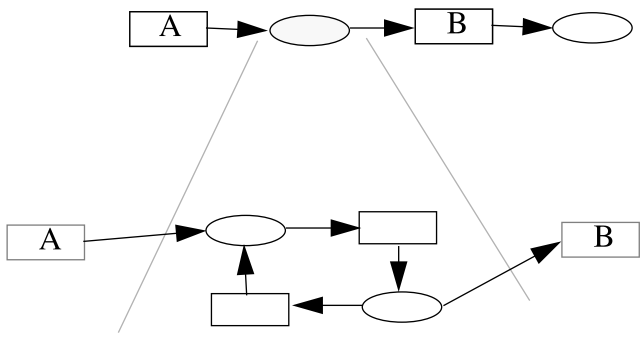


Figure 23. Wrong refinements of composed inference actions.

Additional properties of inference actions can be represented graphically:

- If an inference action has several input stores, does it work if at least one input store is not empty, i.e. not filled with {∅}, or does it work only if all input stores are not empty?
- If an inference action has several output stores does it always fill all of them or only some?

The two questions are likely to arise when dealing with composed inference actions. [Kun89] introduces modelling primitives answering these questions in the domain of dataflow diagrams. These modelling primitives permit to express whether the inputs of an inference action are connected by “and”, “or”, or “exclusive or”. The same can be expressed for the outputs. This notation can be nested to connect several inputs or outputs. Figure 24 shows some examples of how the input of an inference action can be connected. The same can be done for the output of an inference action.

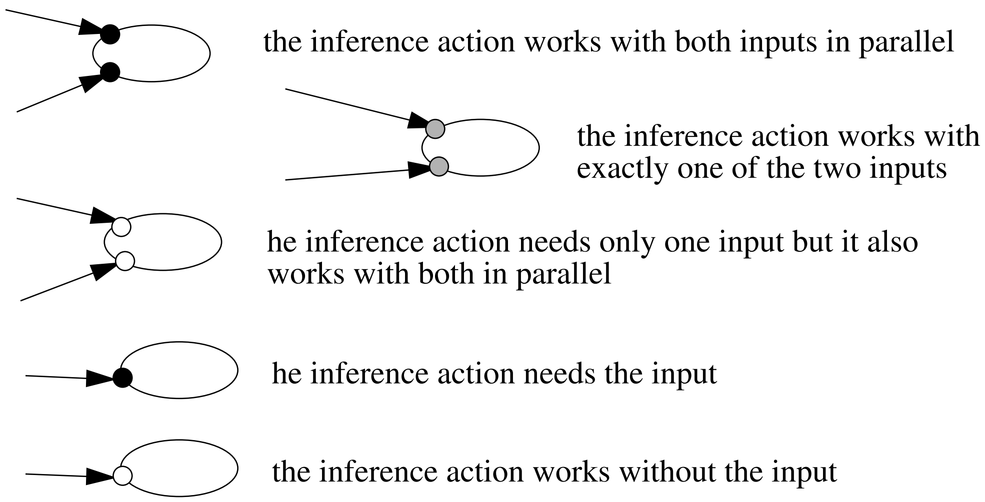


Figure 24. Interface modelling of inference actions (cf. [Kun89])

An elementary inference action *merge* which should compute tuples of elements of two stores only works if each of the two stores contains at least one element. An elementary inference action like *join*, however, which should join all elements of two stores into one set could also work if only one store contains some elements. The graphical descriptions of the cases just mentioned (○ ●) do not have a direct semantics in KARL. The specification of a (composed or elementary) inference action must specify this kind of behavior. Apart from that, these symbols seem to be a meaningful extension of the graphical representation of the inference layer as proposed by [WSB92]. An "or"-connection of two input roles "a" and "b" of an inference action can be expressed by two Horn clauses:

$\forall x,\ldots\ (\ldots \leftarrow x\ \varepsilon\ a \ldots).$
$\forall x,\ldots\ (\ldots \leftarrow x\ \varepsilon\ b \ldots).$

An "and-connection" of two input roles "a" and "b" can be expressed by a Horn clause:

$\forall x,\ldots\ (\ldots \leftarrow x\ \varepsilon\ a\ \wedge x\ \varepsilon\ b \ldots).$

4.3 Task Layer

There are several alternatives for the graphical representation of the procedural knowledge at the task layer. Two well-known means are:

- Nassi-Shneiderman diagrams (cf. [NaS73])
- Program-flow diagrams (cf. [Din66001])

Nassi-Shneiderman diagrams enforce structured programming and are well-suited for hierarchical refinement. A graphical knowledge acquisition tool for them, however, is quite complicated to realize, and the revision of modelled knowledge is quite difficult for the knowledge engineer. Therefore, because of the cyclic manner of knowledge acquisition and the high amount of revision, a variant of program flow diagrams for the task layer is used here. In figure 25, the graphical primitives at the task layer are given:

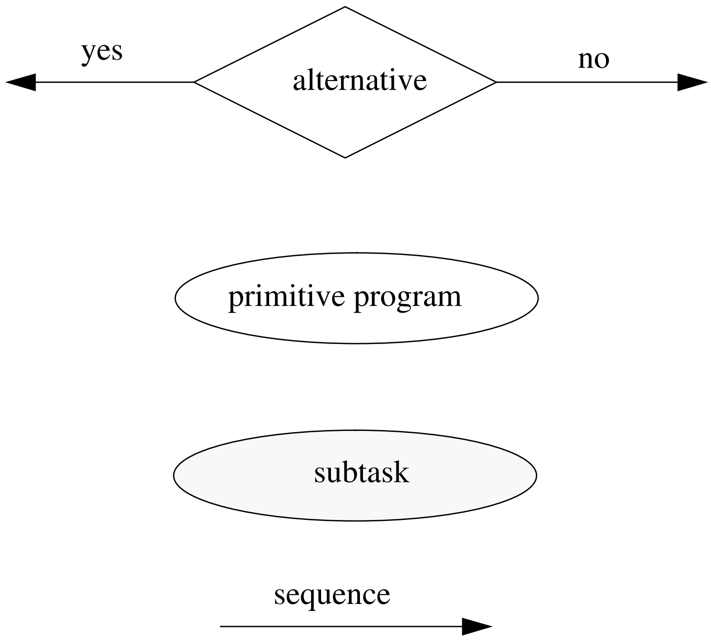


Figure 25. Graphical representation at the task layer.

Currently, there exists no tool support to enforce structured programming with program flow diagrams. But this would be possible.

“the proof of the pudding is in the eating”

Chapter 5

An Example: The Board-Game Method Modelled in KARL[1]

This chapter shows an application of KARL. The two methods chronological backtracking and its task-specific refinement by the board-game method are specified in KARL. The two methods are compared with the solution of the Sisyphusproblem in chapter three, and their mappings for this domain are discussed. Finally, some general conclusions are drawn as to how weak problem-solving methods can be refined to “stronger” task-specific problem solving methods, and how their description in KARL can assist in describing such methods.

5.1 Chronological Backtracking

In chapter 3, a ***beam*** search (cf. [Bun90]) is modelled which solves the assignment problem of the Sisyphus tasks. A short characterization of this problem-solving method is given by (see figure 26) by the following items:

- (1) Check if some of the given states fulfil the goal predicate.
- (2) If they do not, derive the successors of the states in question.
- (3) Over-write the states by the successors.

---

1. I thank Henrik Eriksson for our fruitful private discussions about chronological backtracking and the board-game method. Without these discussions the specification of these methods in KARL as done in this chapter would not have been possible.

- Repeat at (1).

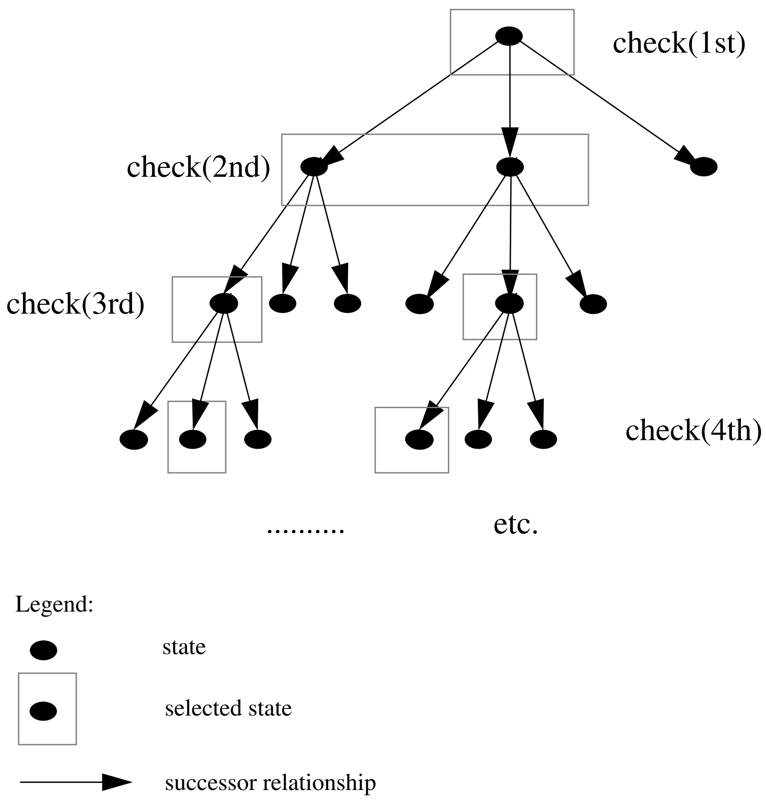


Figure 26. Beam search (branching factor m = 2)

Beam-search is a generalization of ***hill-climbing*** (cf. [Bun90]). Hill-climbing choses only one state as successor state. Instead of considering only one state, beam-search examines a few states and their successors. Non of these two search strategies can guarantee that they find a solution even if one exists. An alternative is ***depth-first search*** or ***chronological backtracking*** (cf. [Bun90]), which is discussed according to [EST92]. A short characterization of it is given by (see figure 27) the following items:

- (1) Check whether the given state fulfils the goal predicate.
- (2) If it does not derive all its successors.
- (3) Delete successors if they are equal to already considered states.
- (4) If no successor exists backtrack until you find a state which has not been already considered and go to (1).
- (5) If a successor exists, select one of the successors as new state.
- Go back to (1) and repeat.

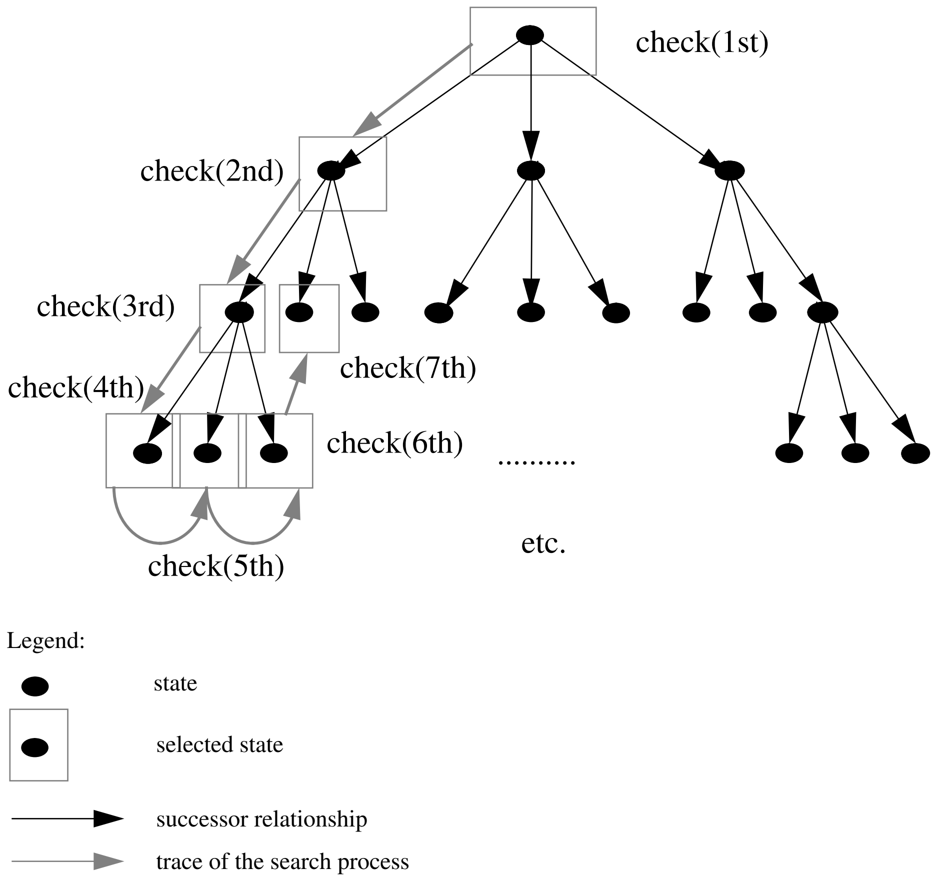


Figure 27. Depth-first search

In a similar way as hill-climbing can be generalized to beam search, depth-first search can be generalized to ***beam search with chronological backtracking*** by considering several alternatives in parallel. Not necessarily all nodes and not necessarily one node, but a number of nodes of the same level are considered and expanded (see figure 28).

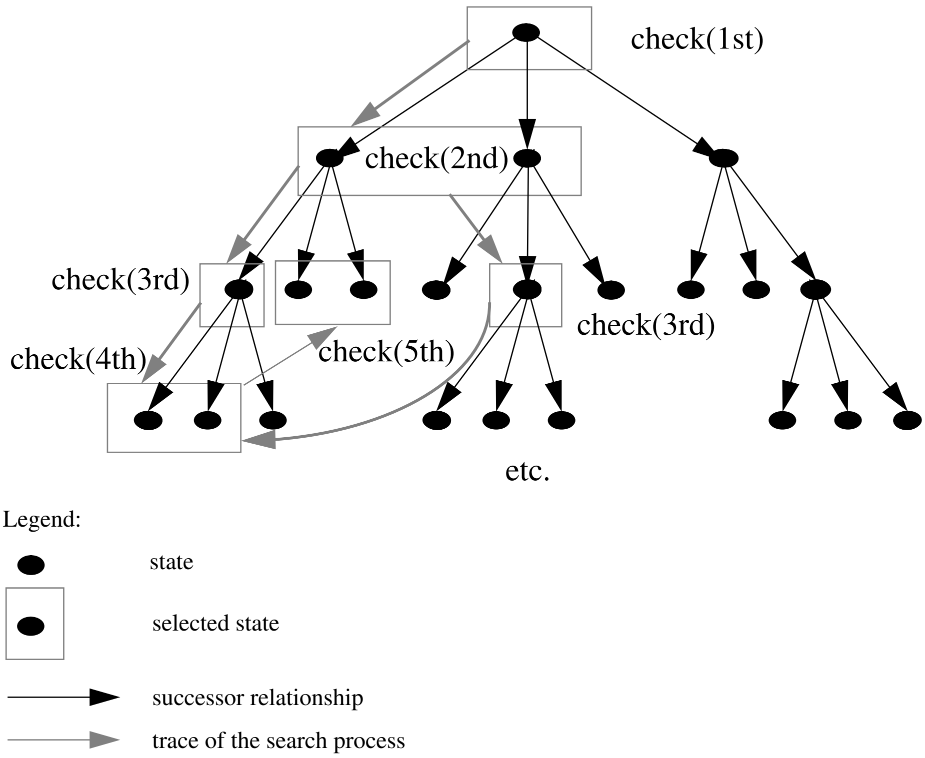


Figure 28. Beam search (branching factor m = 2) with backtracking

State-space search is a fundamental artificial-intelligence technique for problem solving. Tasks can be modelled in terms of problem ***states***, and the problem-solving behavior patterns can be encoded as a search through a state space. The ***chronological-backtracking*** method searches for a sequence of states where the ***initial*** states and ***goal*** states are given, and where two consecutive states in the sequence satisfy constraints on how states can follow one another. The concept of state transitions is important for state-space search. For the chronological-backtracking method, [EST92] uses the notion of a ***transition function*** (T-function) that produces a set of subsequent states of the current state. To reuse the chronological-backtracking method for new tasks, the developer must define the initial states and goal states (or a goal predicate) and the T-function.

The following demonstrates how chronological backtracking or its extension to beam search with chronological backtracking is modelled in KARL and gives a comparison with the beam search for the Sisyphus room-assignment problem modelled in chapter 3. As already mentioned, chronological-backtracking method can use ***depth-first search*** or ***beam search***, where the method considers several alternatives at the same search level in parallel. Whether beam search with backtracking or chronological backtracking is modelled depends on the predicate

priority(less : X, high : Y)

which is used in the elementary inference action *Select_States* (cf. section 5.1.2.2). If the predicate *priority* defines a total ordering on the states, only one state is checked if it is a goal state, and only for this state the successors are derived. If the predicate *priority* defines a partial ordering then a few states are checked in parallel if any of them is a goal state, and the states´ successors are derived. The *priority* predicate is defined in the domain view of the elementary inference action *Select_States*. In domains where sufficient knowledge is given to identify exactly one best successor state, only this state is considered in the next step, and beam search collapses to depth-first search. In domains where the knowledge possibly delivers more than one best state, all states which cannot be distinguished by the domain heuristics are considered in parallel.

The alphabets of the KARL models are not presented because they are assumed implicitly given by the terms used for the inference layer and task layer. The inference layer and task layer are shown in parallel for every level of refinement to improve the readability of the specification.

5.1.1 The Top-Level Inference Action *Chronological_Backtracking*, i.e. the Task *Chronological_Backtracking*

In this subsection, task layer and inference layer of chronological backtracking are shown at the lowest level of refinement.

5.1.1.1 The Entire Task *Chronological_Backtracking*

First, the store variables and the output-data part are initialized. Then the task *Chronological_Backtracking* is called (see figure 29).

```
init;
Chronological_Backtracking()
```

Figure 29. The entire task *Chronological_Backtracking*

5.1.1.2 The Top-Level Inference Action *Chronological_Backtracking*

The composed inference action *Chronological_Backtracking* is defined in figure 30.

```
TOP LEVEL INFERENCE ACTION Chronological_Backtracking
    ELEMENTARY INFERENCE ACTIONS
        Init_States;
    COMPOSED INFERENCE ACTIONS
        Bookkeeping, Derive_Successors, Find_Solution;
    STORES
        Old_States, Selected_States, States;
END;
```

Figure 30. The composed inference action *Chronological_Backtracking*

5.1.2 Refinement of *Chronological_Backtracking*

In the following, the task layer and inference layer of chronological backtracking are shown at the first level of refinement.

5.1.2.1 Refinement of the Subtask *Chronological_Backtracking*

The subtask *Chronological_Backtracking* (see figure 31) consists of three subtasks that are repeated until a solution has been found, that is, until the output-data part of the domain layer is not empty, or until no further successor states which are different from an already considered states can be found (i.e., the store *Selected_States* is empty). The three subtasks are:

- *Find_Solution*, which checks the store *Selected_States* whether it contains a goal state. If it does, the best state is stored at the output-data part of the domain layer.
- *Derive_Successor*, which derives the successors of all selected states and checks if they are legal states.
- *Bookkeeping*, which checks whether these successors are different from already considered states. If new states are found some of them are put into the store *Selected_States*.

In addition, *Init_States* derives the initial states.

```
SUBTASK Chronological_Backtracking
    BODY
        (STORES : Selected_States, Old_States) := Init_States(VIEWS : Init_View),
    REPEAT
        /* Is there a selected state which could be used as a solution? */
        Find_Solution(INPUT_STORES: Selected_States);
        IF ∅(o,[1] domain_class_name)[2]
        THEN
            /* Find_Solution did not find a solution. Therefore, new states are computed
            which could be a possible solution. */
            Derive_Successors(INPUT_STORES: Selected_States; OUTPUT_STORES: States);
            /* Some of the successors states are selected as new selected states. */
            Bookkeeping(INPUT_STORES: States; OUTPUT_STORES: Selected_States);
        ENDIF
    UNTIL (¬∅(o, domain_class_name) ∨ ∅(Selected_States, selected_states))
ENDSUBTASK
```

Figure 31. The subtask *Chronological_Backtracking.*

1. The variable “o” denotes the output-data part of the domain layer.
2. The formula ∅(o, domain_class_name) is evaluated to true if the output-data part “o” does not contain an element of the class *domain_class_name*.

5.1.2.2 Refinement of the Composed Inference Action *Chronological_Backtracking*

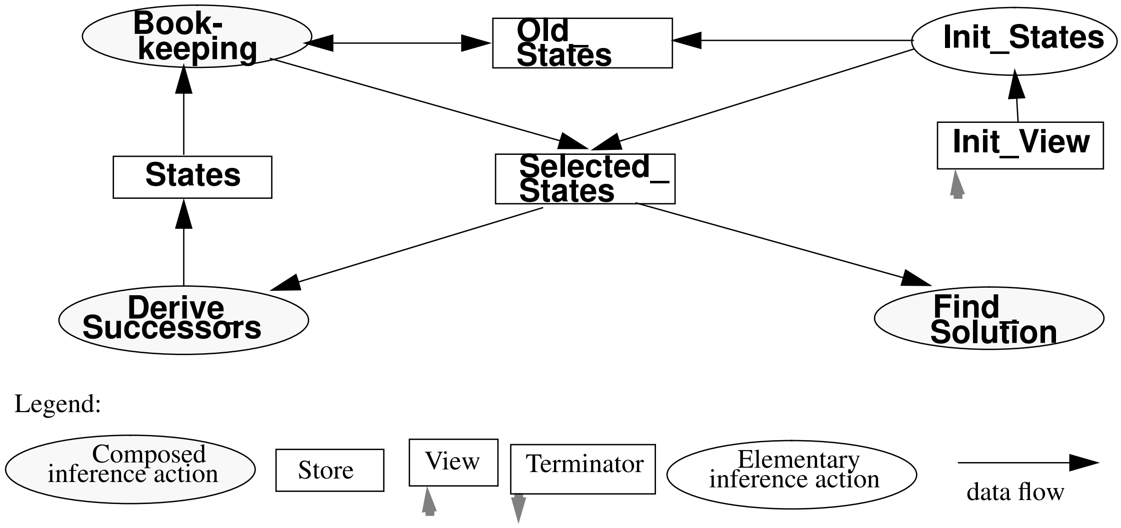


Figure 32. Inference structure of *Chronological_Backtracking*

The store *Selected_States* contains the currently considered states. The store *States* contains the successors of the store *Selected_States*. The store *Old_States* memorizes all considered states. The composed inference action *Bookkeeping* will use this store as input to select some of its elements as new considered states. The view *Init_View* delivers the initial states. These stores and the view are defined in figure 33.

```
STORE Old_States
    DEFINITIONS
        CLASS old_states
            ELEMENT_ATT
                    number : {Integer};[1]
        END;
END;

STORE Selected_States
    DEFINITIONS
        CLASS selected_states
            ELEMENT_ATT
                    number : {Integer};
        END;
END;

STORE States
    DEFINITIONS
        CLASS states
            ELEMENT_ATT
                    number : {Integer};
        END;
END;

VIEW Init_View
    DEFINITIONS
```

1. The attribute *number* is used to formalize the level of a state in the search tree of chronological backtracking. This attribute value is used to ensure that only the states of the currently lowest level in the search tree are selected as elements of *Selected_States* by the composed inference action *Bookkeeping*.

```
        CLASS initial_states
        END;
    END;
```

Figure 33. Stores and view definitions of the composed inference action *Chronological_backtracking*

The elementary inference action *Init_States* (see figure 34) creates the initial states. These states are written to the store *Selected_States* and *Old_States* with the number attribute set to zero. *Init_States* uses the view *Init_View* to read the necessary knowledge from a domain layer.

```
ELEMENTARY INFERENCE ACTION Init_States
    PREMISES
        Init_View;
    CONCLUSIONS
        Old_States, Selected_States;
    RULES
        ∀x_St (x_St [number : 0] ε selected_states ← x_St ε initial_states ).
        ∀x_St(x_St[number : 0] ε old_states ← x_St ε initial_states ).
END
```

Figure 34. The elementary inference action *Init_States*

The composed inference action *Bookkeeping* designates some states as selected states. These states are not equal to already considered ones. The composed inference action *Derive_Successors* creates the successor states of all selected states, and the composed inference action *Find_Solution* checks if the selected states are a solution. If they are, the best is stored at the output-data part. These three composed inference actions are specified in figure 35.

```
COMPOSED INFERENCE ACTION Bookkeeping
    PREMISES
        Old_States, States;
    CONCLUSIONS
        Old_States, Selected_States;
    ELEMENTARY INFERENCE ACTIONS
        Copy, Select _States;
    VIEWS
        Select_States_Relation;
END;

COMPOSED INFERENCE ACTION Derive_Successors
    PREMISES
        Selected_States;
    CONCLUSIONS
        States;
    ELEMENTARY INFERENCE ACTIONS
        Create, Prune;
    VIEWS
        Correct_States, Successor_Relation;
    STORES
        Successors;
END;

COMPOSED INFERENCE ACTION Find_Solution
    PREMISES
```

```
        Selected_States;
    ELEMENTARY INFERENCE ACTIONS
        Check, Select_Solution;
    VIEWS
        Goal_States, Select_Solution_Relation;
    STORES
        Possible_Solutions;
    TERMINATORS
        Solution;
END;
```

Figure 35. The composed inference actions *Bookkeeping*, *Derive_Successors*, and *Find_Solution*.

5.1.3 Refinement of *Bookkeeping*

Bookkeeping selects some states as selected states. These states are different from already considered ones.

5.1.3.1 Refinement of the Subtask *Bookkeeping*

```
SUBTASK Bookkeeping
    INPUT_STORES Old_States, States
    OUTPUT_STORES Old_States, Selected_States
    BODY
        /* Some states are selected according to the priority predicate. */
        (STORES: Selected_States) :=
            Select_States(STORES: Old_States, States; VIEWS: Select_States_Relation);
        /* The selected states are stored as already considered ones. */
        (STORES: Old_States) := Copy(STORES: Selected_States);
ENDSUBTASK
```

Figure 36. The subtask *Bookkeeping*

5.1.3.2 Refinement of the Composed Inference Action *Bookkeeping*

The composed inference action *Bookkeeping* (see figure 37) checks if states are different from already considered states. If new states are found, the best of them are put into the store *Selected_States*.

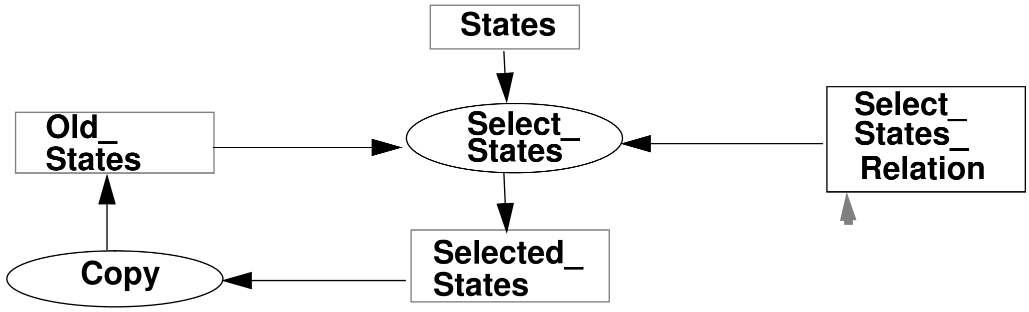


Figure 37. Refinement of *Bookkeeping*

The domain view *Select_States_Relation* provides the domain knowledge required to identify equal states, and to define a priority for the states.

```
VIEW Select_States_Relation
    DEFINITIONS
        PREDICATE equal
                e1 : {states};
                e2 : {old_states};
        END;
        PREDICATE priority
                less : {states};
                high : {states};
        END;
END;
```

Figure 38. The definition of the view *Select_States_Relation*

The elementary inference action *Copy* copies all selected states to *Old_States* to prevent that a state is considered as selected for a second time. The elementary inference action *Select_States* selects the states to be considered next.

/* *Copy* copies all selected states to old states */
ELEMENTARY INFERENCE ACTION ***Copy***
PREMISES
Selected_States;
CONCLUSIONS
Old_States;
RULES
/* Every selected state is a new element of old states. */
$\forall x_{St}(x_{St} \in old\_states \leftarrow x_{St} \in selected\_states\ ).$
END;

/* *Select_States* selects the states to be considered next. */
ELEMENTARY INFERENCE ACTION ***Select_States***
PREMISES
Old_States, Select_States_Relation, States;
CONCLUSIONS
Selected_States;
DEFINITIONS
CLASS ***not_used_states***
ISA ***states;***
END;
RULES

/* A state is not selected, i.e. it is assigned to the class *not_used_states*, if another state (that is not equal to an element of *Old_States*) is at a lower level of recursion. */
$\forall x_{St}, \forall y_{St}, \forall z_{St}, \forall x_n, \forall y_n$
$(x_{St}$ ε ***not_used_states*** $\leftarrow$
x_{St} ***[number :*** x_n***]*** ε ***states*** $\wedge$
y_{St} ***[number :*** y_n***]*** ε ***states*** $\wedge$
z_{St} ε ***old_states*** $\wedge$
$x_n < y_n \wedge$
$\neg$***equal(e1 :*** y_{St}***, e2 :*** z_{St}***).***
/* A state is not selected if another state (that is not equal to an element of *Old_States*) at the same level of recursion has a higher priority. */
$\forall x_{St}, \forall y_{St}, \forall z_{St}, \forall x_n,$
$(x_{St}$ ε ***not_used_states*** $\leftarrow$
x_{St} ***[number :*** x_n***]*** ε ***states*** $\wedge$
y_{St} ***[number :*** x_n***]*** ε ***states*** $\wedge$
priority(less : x_{St}***, high :*** y_{St}) $\wedge$
z_{St} ε ***old_states*** $\wedge$
$\neg$***equal(e1 :*** y_{St}***, e2 :*** z_{St}***).***
/* A state is selected if it is not an element of the classes *not_used_states* and not equal to an element of *Old_States*. */
$\forall x_{St}, \forall z_{St}$ $(x_{St}$ ε ***selected_states*** $\leftarrow$
x_{St} ε ***states*** $\wedge$
$\neg x_{St}$ ε ***not_used_states*** $\wedge$
z_{St} ε ***old_states*** $\wedge$
$\neg$***equal(e1 :*** x_{St}***, e2 :*** z_{St}***)).***
END;

Figure 39. The elementary inference actions *Copy* and *Select_State*.

5.1.4 Refinement of *Derive_Successors*

Derive_Successors derives the correct successors of the selected states.

5.1.4.1 Refinement of the Subtask *Derive_Successors*

SUBTASK ***Derive_Successors***
INPUT_STORES ***Selected_States***
OUTPUT_STORES ***States***
BODY
/* Successor states are created for the selected states. */
(STORES: Successors) :=
Create(STORES: Selected_States; VIEWS: Successor_Relation)
/* The successor states are pruned. The good ones are put into the basket store. */
(STORES: States) := Prune(STORES: Successors; VIEWS: Correct_States)
ENDSUBTASK

Figure 40. The subtask *Derive_Successors*

5.1.4.2 Refinement of the Composed Inference Action *Derive_Successors*

The composed inference action *Derive_Successors* (see figure 41) derives the successors of all selected states, and checks if these successors are legal states.

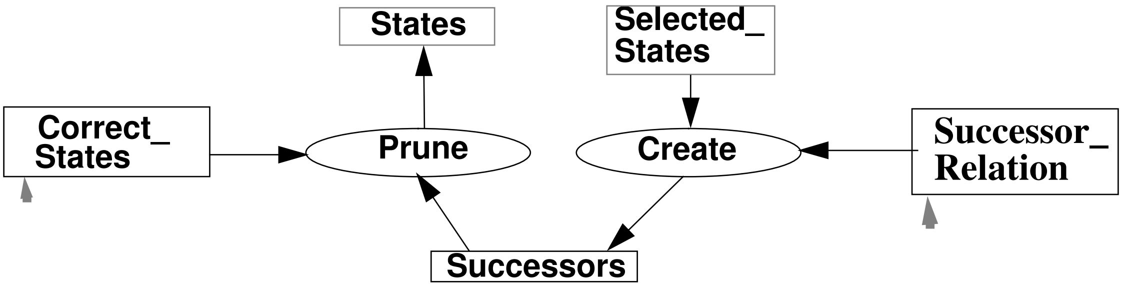


Figure 41. Refinement of *Derive_Successors*

The store *Successors* contains the current successor states of the states in the store *Selected_States*. The definitions of the store *Successors* are similar to the definition of the store *States*. The view *Successor_Relation* is used to define the successor relationship (i.e., the *T_function*) of states. The view *Correct_States* must define when a state is a legal state.

```
STORE Successors
    DEFINITIONS
        CLASS successors
            ELEMENT_ATT
                    number : {Integer};
        END;
END;

VIEW Corrcect_States
    DEFINITIONS
    PREDICATE correct_states
                    s : {successors};
        END;
END;

VIEW Successor_Relation
    DEFINITIONS
        PREDICATE T_function
                old : {selected_states};
                new : {successors};
        END;
END;
```

Figure 42. The definition of the store *Successors* and the views *Correct_States* and *Successor_Relation*.

The elementary inference action *Create* (see figure 43) creates all successor states from one or more states in the store *Selected_States*. The successors are written to the store *Successors*. *Create* uses the view *Successor_Relation* to read the required knowledge from a domain layer. The view must define the relation *T_function*. The elementary inference action *Prune* (see figure 43) checks if successor states are legal. If yes, they are considered as new states and written to the store *States*. The view *Correct_States* must define the predicate *correct_states*.

```
ELEMENTARY INFERENCE ACTION Create
    PREMISES
        Selected_States, Successor_Relation;
```

```
    CONCLUSIONS
        Successors;
    RULES
        /* An element of the class successor is a new state that is a successor of a selected state. */
        ∀x_St, ∀y_St, ∀x_n, ∀y_n
            (y_St [number : y_n] ε successors ←
                x_St [number : x_n] ε selected_states ∧
                T_function(old : x_St, new :y_St) ∧
                +(x_n,1,y_n) ).[1]
END
ELEMENTARY INFERENCE ACTION Prune
    PREMISES
        Correct_States, Successors;
    CONCLUSIONS
        States;
    RULES
        /* Every successor state which is correct is a new element of the class states. */
        ∀x_St (x_St ε states ← x_St ε successors ∧ x_St ε correct_states(s : x_St)).
END;
```

Figure 43. The elementary inference actions *Create* and *Prune*.

5.1.5 Refinement of *Find_Solution*

Find_Solution checks the selected states whether they are a solution. If they are, the best is stored at the output-data part.

1. One is added to the attribute value number because the successor state is one lower in the search tree than the original state.

5.1.5.1 Refinement of the Subtask *Find_Solution*

SUBTASK ***Find_Solution***
  INPUT_STORES ***Selected_States***
  BODY
    /* The selected states are checked whether they contain a possible solution. */
    (STORES: Possible_Solutions) :=
      Check(STORES: Selected_States; VIEWS: Goal_States)
    /* A solution is selected. */
    (TERMINATOR: Solution) :=
      Select_Solution(STORES: Possible_Solutions;
        VIEWS: Select_Solution_Relation)
ENDSUBTASK

Figure 44. The subtask *Find_Solution.*

5.1.5.2 Refinement of the Composed Inference Action *Find_Solution*

The composed inference action *Find_Solution* checks if the selected states are a solution. If they are, the best is stored at the output-data part.

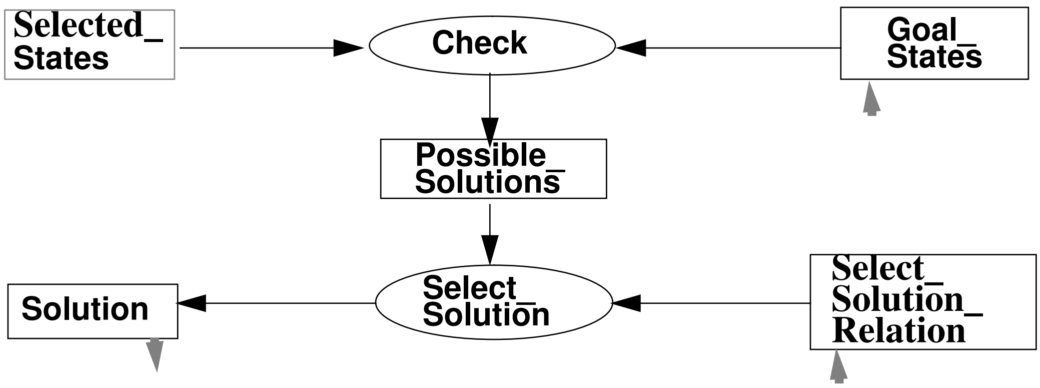


Figure 45. Refinement of *Find_Solution.*

The store *Possible_Solutions* contains all possible solutions. The terminator *Solution* writes one selected solution on the output-data part of the domain layer. The view *Goal_States* is used by the elementary inference action *Check* to determine whether a state is a possible solution. The view *Select_Solution_Relation* is used by the elementary inference action *Select_Solution* to select one solution out of all possible ones.

STORE ***Possible_Solutions***
  DEFINITIONS
    CLASS ***possible_solutions***
    END;
END;
TERMINATOR ***Solution***
  DEFINITIONS
    CLASS ***solution***
    END;
END;

```
VIEW Goal_States
    DEFINITIONS
        PREDICATE goal
                    g : {selected_states};
        END;
END;
VIEW Select_Solution_Relation
    DEFINITIONS
        PREDICATE priority_solutions
                    less : {possible_solutions};
                    high : {possible_solutions};
        END;
END;
```

Figure 46. The stores, terminators, and views of *Find_Solution*.

The elementary inference action *Check* checks whether one or some states state are possible solutions. The elementary inference action *Select_Solution* selects one solution from all possible ones.

```
ELEMENTARY INFERENCE ACTION Check
    PREMISES
        Goal_States, Selected_States;
    CONCLUSIONS
        Possible_Solutions;
    RULES
        /* Every state which fits the goal predicate is a possible solution. */
        ∀x_St
            (x_St ε possible_solutions ←
                    x_St ε selected_states ∧
                    goal(g : x_St)).
END;
ELEMENTARY INFERENCE ACTION Select_Solution
    PREMISES
        Possible_Solutions;
    CONCLUSIONS
        Solution;
    DEFINITIONS
        CLASS highest_priority
            ISA possible_solutions;
        END;
    RULES
        /* The elements of all possible_solutions with the highest priority are selected. */
        ∀x_St, ∀y_St (x_St ε highest_priority ←
            x_St ε possible_solutions ∧
            y_St ε possible_solutions ∧
            ¬ priority_solutions(less : x_St, high : y_St)).
        /* One element is selected. */
        ∀x_St, ∀y_St (x_St ε solution ←
            x_St ε highest_priority ∧ y_St ε highest_priority ∧
            ¬ x_St< y_St).
END;
```

Figure 47. The elementary inference actions *Check* and *Select_Solution*.

5.1.6 A Comparison of *Chronological_Backtracking* and the Solution of the Sisyphus Room-Assignment Problem in Chapter Three

Three main comments can be made on comparing chronological backtracking (or beam search with backtracking) with the solution of the Sisyphus task (called ***Sisyphusmethod*** further on) as discussed in chapter 3.

It is necessary to introduce two new elementary inference actions *Copy* and *Select_States* which manage the bookkeeping of chronological backtracking, i.e. they help to decide whether states are already used or not. An attribute *number* is introduced for states to represent the level that a state has in the search tree. As, at this point, the complete inference structure contains seven elementary inference actions and several roles, it is split up into three separate composed inference actions. The inference action *Bookkeeping* manages the bookkeeping of chronological backtracking, the composed inference action *Derive_Successors* contains the two steps of creating and pruning successors, and the composed inference action *Find_Solution* contains the two steps of checking and selecting a solution. Only the elementary inference action *Init_States* remains as an element of the highest level of refinement. The two views *Next_Comp* and *Next_Slots* of the Sisyphusmethod merge into one view *Successor_Relation* because chronological backtracking makes no assumption about the internal structure of states and successor states (i.e., it does not distinguish between components and slots). A new view *Init_view* is defined, because the generic problem-solving method of chronological backtracking cannot hard-wire the initial states at the inference layer as done by the elementary inference action *Init* of the Sisyphusmethod.

The mapping parts of the views and the terminator of chronological backtracking are empty. This is no surprise because in chapter three a problem-solving process connected with a specific domain is modelled, whereas chronological backtracking is a generic problem-solving method without any connection to a domain. If it were intended to solve Sisyphus with chronological backtracking it would be necessary to define the mapping rules and definitions in a similar way for it. But that would still not be enough: Definition parts and bodies of the elementary inference actions and definition parts of the stores in ***chronological backtracking*** are shorter than those for the Sisyphusmethod. The formal specification of the elementary inference action is, in most cases, one simple clause. Also, the terminological knowledge is trivial. This implies that, if chronological backtracking is to be applied for a task similar to the Sisyphus problem (i.e., for assignment problems), new definitions and clauses ***at the inference layer*** (i.e., in ***stores*** and ***elementary inference actions***) have to be introduced. Above all, the terminological knowledge of the internal structure of a state has to be defined, i.e. that a state is a set of assignments which are tuples of slots and components. In addition, further clauses must be specified. For example, how a successor state is derived from a state and that the successor inherits assignments of the old state.

This result surprised us, because we supposed that generic problem-solving methods such as chronological backtracking could be adapted to domains by merely specifying mapping rules and definitions. Otherwise, the fact that a state has a certain internal structure is not really a domain dependent circumstance. It is ***domain*** dependent that slots are working places instead of gates and that components are employees instead of airplanes. These are the things which are mapped by the mapping clauses. The fact that a state is a set of assignments, however, is related to the ***task*** which should be solved with the problem-solving method. Thus, the adaptation of chronological backtracking to the Sisyphus room-assignment problem implies two different activities.

- First, the generic problem-solving method chronological backtracking (also called a ***weak*** method) has to be instantiated to a so-called ***strong*** problem-solving method (cf. [McD88]), which makes specific, i.e. strong, ***assumptions about the task*** which should be solved by this method.
- Then, this task-specific instantiation of chronological backtracking has to be mapped to the domain of the room-assignment problem.

5.2 Board-Game Method

> "The idea of the board-game method is to view the problem as a board game where ***pieces*** can be moved between ***locations***. We assume that the game has a fixed number of pieces and locations. More than one piece can be moved to the same location simultaneously, and, if required, the pieces at each location can be ordered. ... We have implemented the board-game method using the chronological backtracking method." [EST92], p. 16.

The board-game method is generic in the sense that the developer can ***configure*** the method to perform a number of game-like tasks. For instance, [EST92] have applied the board-game method to the Towers-of-Hanoi game, the cannibals-and-missionaries game, and the Sisyphus room-assignment problem. To configure the board-game method to support a new board-game task, the developer must identify the pieces and locations of the game, and define the constraints under which pieces can move. For example, the rules of the game might stipulate that the target location for a move must be empty. Also, the board-game method requires an initial state and goal states of the game (or a goal predicate that is satisfied when the goal is reached). In addition to constraints on moves, certain games have constraints in terms of "illegal" game states.

The board-game method is a refinement of chronological backtracking. ***The refinement is not accomplished by introducing additional inference actions, additional levels of refinement, or by changing the task layer, but this refinement is achieved by adding terminological knowledge and new clauses to roles and elementary inference actions.*** The following section does not deal with the whole specification of the board-game method, but only the extension of the chronological backtracking method by the board-game method.

5.2.1 Knowledge About the Terminological Structure of the Task

New class definitions and predicate definitions and new attributes in existing class definitions are introduced.

Three new class definitions are introduced in all stores. These class definitions are a means of defining the internal structure of a state:

CLASS ***assignments***
ELEMENT_ATT
location : {***locations***};
order : {***Integer***};
piece : {***pieces***};
END;
CLASS ***pieces***
END;
CLASS ***locations***
END;

An assignment is a triple of a location, a piece, and an ordering of pieces. Information pertaining to order is relevant in domains such as the Towers-of-Hanoi game, where several pieces have the same location but differ in their ordering.

All other original class definitions in roles and elementary inference actions of chronological backtracking are extended by the set-valued attribute *assign*:

assign :: {***assignments***}

Thus, the definition of a state looks like:

```
CLASS states
    ELEMENT_ATT
        assign :: {assignments};
        number : {Integer};
END;
```

The attribute *assign* is used to define the internal structure of a state. It is characterized by a set of assignments.

The class *moves* is introduced which is used to specify the predicate *T_function* applied in the elementary inference action *Create*:

```
    CLASS moves
    ELEMENT_ATT
        actions :: {assignments};
        assumpt :: {assignments};
END;
```

The attribute *assumpt* describes the conditions that must be fulfilled by a state such that a move can be applied to it. The attribute *actions* describes the modified assignments as a result of applying the move to a state.

The new predicate *other_assignments* is introduced which is used in the elementary inference action *Select_States* for detecting equal states, i.e. states which have already been considered. Two states are equal if they have the same set of assignments. Chronological backtracking makes no assertion about when two states are equal. The predicates *apply* and *not_apply* model tuples of selected states and moves when the move can or cannot be applied to the state in question. The predicate *not_changed_pieces* is an auxiliary predicate that distinguishes the pieces which change location from the pieces that do not change location in a move.

```
PREDICATE apply
    ELEMENT_ATT
        s : {selected_states};
        m : {moves};
END;
PREDICATE other_assignment
    state : {};
    has_other_assign : {};
END;
PREDICATE not_apply
    ELEMENT_ATT
        s : {selected_states};
        m : {moves};
END;
PREDICATE not_changed_pieces
    ELEMENT_ATT
        state : {selected_states};
        move :{moves};
        pieces : {pieces};
END;
```

5.2.2 Knowledge About the Logical Description of Inferences Related to the Task

Additional knowledge about the inferences is added to the body of the elementary inference actions *Create* and *Select_States*. The new terminology is used to define in more detail how to derive a successor state.

In *Create*, a successor state is defined by the elements of the class *moves*. A move is a tuple of two sets, a set of actions and a set of assumptions. An action derives a new location for a piece. Therefore, if the board-game method is adapted to a specific domain, a move must be defined instead of the general *T_function* in chronological backtracking in the upward mapping of the view *Successor_Relation*. The class *assumptions* models the assumptions of a move. The entire description of the elementary inference action is shown (with the newly introduced parts marked) in figure 48.

ELEMENTARY INFERENCE ACTION Create
PREMISES
Selected_States, Successor_Relation;
CONCLUSIONS
Successors;
DEFINITIONS
CLASS moves
ELEMENT_ATT
actions:: {assignments };
assumpt :: {assignments};
END;
PREDICATE apply
ELEMENT_ATT
s : {selected_states};
m :{moves};
END;
PREDICATE not_apply
ELEMENT_ATT
s : {selected_states};
m :{moves};
END;
PREDICATE not_changed_pieces
ELEMENT_ATT
state : {selected_states};
move :{moves};
pieces : {pieces};
END;
RULES
/* An element of the class successors is a state which is a successor of a selected state. */
$\forall x_{St}, \forall y_{St}, \forall x_n, \forall y_n$
(y_{St} [number : y_n] ε successors ←
x_{St} [number : x_n] ε selected_states ∧
T_function(old : x_{St}, new :y_{St}) ∧
+(x_n,1,y_n)).
/* A move defines a successor of a selected state if its assumptions are a
subset of the assignments of this state. */
$\forall \boldsymbol{x_m}, \forall \boldsymbol{x_{St}}$

(T_function(old : x_{St}, new : NewSt(x_{St},x_m))[1] ←
x_m ε moves ∧
x_{St} ε selected_states ∧
apply(state : x_{St}, move : x_m)).
∀x_A, ∀x_m, ∀x_{St}
(apply(state : x_{St}, move : x_m) ←
x_m ε moves ∧
x_{St} ε selected_states ∧
¬not_apply(state : x_{St}, move : x_m)).
∀x_A, ∀x_m, ∀x_{St}
(not_apply(state : x_{St}, move : x_m) ←
x_m[assumpt :: {x_A}] ε moves ∧
x_{St} ε selected_states ∧ ***¬x_{St}[assign :: {x_A}]).***
/* A successor gets the ***changed*** assignments. */
∀x_A, ∀x_l, ∀x_m, ∀x_p, ∀x_O, ∀x_{St}
(NewSt(x_{St},x_m)[assign :: {x_A[piece : x_p, order : x_O, location : x_l]}] ←
x_m[actions :: {x_A[piece : x_p, order : x_O, location : x_l]}] ε moves ∧
T_function(old : x_{St}, new : NewSt(x_{St},x_m))).
/* A successor gets the ***unchanged*** assignments. An assignment of a piece remains unchanged if the actions of a move do not contain this piece. */
∀x_A, ∀x_p, ∀x_m, ∀x_{St}
NewSt(x_{St},x_m)[assign :: {x_A}] ←
x_m ε moves ∧
x_{St}[assign :: {x_A[piece : x_p]}] ε selected_states ∧
not_changed_pieces(state : x_{St}, move : x_m, pieces :: {x_p}) ∧
T_function(old : x_{St}, new : NewSt(x_{St},x_m))).
∀x_A,∀x_m, ∀x_p, ∀x_{St}
(not_changed_pieces(state : x_{St}, move : x_m, pieces :: {x_p}) ←
x_{St}[assign :: {x_A[piece : x_p]}] ε selected_states ∧
x_m[actions :: {y_A}] ε moves ∧ ***¬y_A[piece : x_p]).***
END;

Figure 48. The elementary inference action *Create*.

The equality of two states is defined in the elementary inference action *Select_States* by the predicate *equal*. Two states are equal, if their sets of assignments are equal. As in chronological backtracking there are no attributes which describe states, it is not possible to define the predicate. The definition of the predicate is domain-independent, i.e. it is not defined in the mapping of the elementary inference action *Select_States* but in its rule part. The entire description of the elementary inference action is shown (with the new introduced parts marked) in figure 49.

/* Select_States selects some states which will be considered next. */
ELEMENTARY INFERENCE ACTION Select_States
PREMISES
Old_States, Select_States_Relation, States;
CONCLUSIONS
Selected_States;

1. The object identity of the new state depends functionally on the object identity of the old state and the object identity of the move.

DEFINITIONS
CLASS not_used_states
ISA states;
ELEMENT_ATT
assign :: {***assignments***};
number : {Integer};
END;
PREDICATE equal
e1 : {states};
e2 : {old_states};
END;
PREDICATE other_assignment
state : {};
has_other_assign : {};
END;
PREDICATE priority
less : {states};
high : {states};
END;
RULES
/* A state is not selected, i.e. it is assigned to the class *not_used_states*, if another state (that is not equal to an element of *Old_States*) is at a lower level of recursion. */
$\forall x_{St}, \forall y_{St}, \forall z_{St}, \forall x_n, \forall y_n$
(x_{St} ε not_used_states ←
x_{St} [number : x_n] ε states ∧
y_{St} [number : y_n] ε states ∧
z_{St} ε old_states ∧
$x_n < y_n$ ∧
¬equal(e1 : y_{St}, e2 : z_{St}).
/* A state is not selected if another state (that is not equal to an element of *Old_States*) at the same level of recursion has a higher priority. */
$\forall x_{St}, \forall y_{St}, \forall z_{St}, \forall x_n$,
(x_{St} ε not_used_states ←
x_{St} [number : x_n] ε states ∧
y_{St} [number : $_{xn}$] ε states ∧
priority(less : x_{St}, high : y_{St}) ∧
z_{St} ε old_states ∧
¬equal(e1 : y_{St} , e2 : z_{St}).
/* A state is selected if it is not an element of the classes *not_used_states* and not equal to an element of *Old_States*.*/
$\forall x_{St}, \forall z_{St}$ (x_{St} ε selected_states ←
x_{St} ε states ∧
¬x_{St} ε not_used_states ∧
z_{St} ε old_states ∧
¬equal(e1 : x_{St}, e2 : z_{St})).
/* A state is equal to an element of *Old_States* if it has the same assignments. */
$\forall \boldsymbol{x_{St}}, \forall \boldsymbol{y_{St}}$
(equal(new : x_{St}, old : y_{St}) ←
$\boldsymbol{x_{St}}$ ε ***states*** ∧
$\boldsymbol{y_{St}}$ ε ***old_states*** ∧
¬***other_assignment(state : x_{St}, has_other_assign : y_{St})*** ∧

$\neg$***other_assignment(state : y_{St}, has_other_assign : x_{St})).***
$\forall x_A$, $\forall x_{St}$, $\forall y_{St}$
(other_assignment(state : x_{St}, has_other_assign : y_{St}) $\leftarrow$
x_{St}[assign :: {x_A}] $\wedge \neg$***y_{St}[assign :: {x_A}]).***
END

Figure 49. The elementary inference action *Select_States*.

5.2.3 The Mapping of the Board-Game Method on the Sisyphus Domain

Chronological backtracking does not only require defining the mapping parts of the views and terminators in order to adapt the problem-solving method to the domain of Sisyphus room-assignment problem, but it also calls for specifying additional knowledge in roles and elementary inference actions. This is different in the board-game method. Due to the fact that this method is designed for tasks in which oieces (i.e., components) are arranged in locations (i.e., slots), the board-game method can be applied to the Sisyphus task without changing the inference layer or task layer. What must be done is defining the mapping from the Sisyphus domain to the inference layer of the board-game method. This mapping is discussed in the following.

The downward mapping from the terminator *Solution* and the upward mapping from the view *Correct_States* are identical to the corresponding mappings in section 3.2.5.3.3.

As no further knowledge is given as how to select a solution out of the possible ones, one solution is selected arbitrarily which must be defined in the upward mapping of the view *Select_Solution_Relation*. This is comparable to the procedure in the body of the elementary inference action *Create* in section 3.2.5.3.4. The heuristics discussed in section 3.0 are previously used in the view *Successor_Relation*. The upward mapping of the view *Select_States_Relation* is empty because of similar reason. If not all legal successors but only one successor state should be regarded further on a total ordering on all states by the built-in predicate “<“ can be defined in this mapping as done in the inference action *Create*.

The upward mapping of the view *Goal_States* checks whether a state assigns every employee to a place. This is done in a quite similar way as in the elementary inference action *Check* of section 3.2.5.3.4.

The upward mapping of the view *Successor_Relation* changes significantly because now a move has to be defined, i.e. sets of actions and assumptions. This mapping is shown in detail in figure 50:

VIEW ***Successor_Relation***
DEFINITIONS
CLASS ***assignments***
ELEMENT_ATT
location : {***locations***};
order : {***Integer***};
piece : {***pieces***};
END;

CLASS ***locations***
END;
CLASS ***moves***
ELEMENT_ATT
actions:: {***assignments***};
assumpt :: {***assignments***};
END;
CLASS ***next_locations***
ISA ***possible_locations;***
END;
CLASS ***next_pieces***
ISA ***possible_pieces;***
END;
CLASS ***pieces***
END;
CLASS ***possible_locations***
ISA ***locations;***
ELEMENT_ATT
states :: {***selected_states***};
END;
CLASS ***possible_pieces***
ISA ***piece***
ELEMENT_ATT
states :: {***selected_states***};
END
PREDICATE ***before_location***
high : {***locations***};
less : {***locations***};
END;
PREDICATE ***before_piece***
high : {***pieces***};
less : {***pieces***};
END;
UPWARD MAPPING
/* 1. The pieces are selected for the next move according to the heuristics of Siggi D. */
/* 1.1. The next pieces to be considered for a state are pieces which have
not already been placed, and which have a priority as high as possible. */
$\forall x_C, \forall x_{St}, \forall y_C$
$(x_C[states :: \{x_{St}\}] \in next\_pieces \leftarrow$
$x_C[states :: \{x_{St}\}] \in possible\_pieces \wedge$
$y_C\,[states :: \{x_{St}\}] \in possible\_pieces \wedge$
$\neg before\_pieces\ (high : y_C, less : x_C))$.
/* 1.2. Possible pieces for a state are pieces which have not
already been placed by the state. */
$\forall x_A, \forall x_C, \forall x_{St}$
$(x_C[states :: \{x_{St}\}] \in possible\_pieces \leftarrow$
$x_{St}[assign :: \{x_A[piece : x_C, location : nil]\}] \in selected\_states \wedge$
$x_C \in pieces)$.
/* 1.3. The priority of the pieces is: head of the group, secretary,
manager, boss, etc. */
$\forall x_C, \forall y_C$
$(before\_piece(high : x_C, less : y_C) \leftarrow$
$\neg(x_C \cong y_C) \wedge x_C \in big\_boss \wedge y_C \in employees).$

$\forall x_C, \forall y_C, \forall y_V$
$(before\_piece(high : x_C, less : y_C) \leftarrow$
$x_C[role : \text{"Secretary"}] \ \varepsilon \ employees \wedge$
$y_C[role : y_V] \ \varepsilon \ employees \wedge$
$\neg(y_V \cong \text{"Secretary"}) \wedge$
$\neg(y_C \ \varepsilon \ big\_boss)).$
$\forall x_C, \forall y_C, \forall y_V$
$(before\_piece(high : x_C, less : y_C) \leftarrow$
$x_C[role : \text{"Manager"}] \ \varepsilon \ employees \wedge$
$y_C[role : y_V] \ \varepsilon \ employees \wedge$
$\neg(y_V \cong \text{"Secretary"}) \wedge$
$\neg(y_V \cong \text{"Manager"}) \wedge$
$\neg(y_C \ \varepsilon \ big\_boss)).$
$\forall x_C, \forall y_C, \forall y_V$
$(before\_piece(high : x_C, less : y_C) \leftarrow$
$x_C \ \varepsilon \ boss \wedge$
$y_C[role : y_V] \ \varepsilon \ employees \wedge$
$\neg(y_C \ \varepsilon \ boss) \wedge$
$\neg(y_V \cong \text{"Secretary"}) \wedge$
$\neg(y_V \cong \text{"Manager"})).$

/* 2. The locations are selected for the next move according to the heuristics of Siggi D.*/

/* 2.1. The next locations to be considered for a state are locations that have not already been occupied, and which are as central as possible. */

$\forall x_S, \forall x_{St}, \forall y_S$
$(x_S \ \varepsilon \ next\_location \ [states :: \{x_{St}\}] \leftarrow$
$x_S \ \varepsilon \ possible\_locations[states : x_{St}] \wedge$
$y_S \ \varepsilon \ possible\_locations[states : x_{St}] \wedge$
$\neg before\_location(high : y_S, less: x_S)).$

/* 2.2. Possible locations for a state are locations that have not already been occupied by the state. */

$\forall x_A, \forall x_C, \forall x_S, \forall x_{St}, \forall y_S$
$(y_S \ \varepsilon \ possible\_locations[states :: \{x_{St}\}] \leftarrow$
$x_{St}[assign :: \{x_A[piece : x_C, location : x_S]\}] \ \varepsilon \ selected\_states \wedge$
$y_S \ \varepsilon \ locations \wedge$
$\neg(y_S \cong x_S) \wedge$
$\neg(y_S \cong nil)).$

/* 2.3. A location comes before another if it is more central than the other. */

$\forall x_S, \forall x_V, \forall y_S, \forall y_V$
$(before\_location(high : x_S, less : y_S) \leftarrow$
$x_S[location : x_V] \ \varepsilon \ place \wedge$
$y_S[location : y_V] \ \varepsilon \ place \wedge$
$x_V < y_V).$

/* 3. The assumptions and the actions of a move are determined. */

$\forall x_A, \forall x_C, \forall x_S, \forall x_{St}$
$(NewM(x_{St}, x_C, x_S)[assumpt :: \{x_A\},$
$actions :: \{NewA(x_C, x_S)[pieces : x_C, location : x_S]\}] \ \varepsilon \ moves \leftarrow$
$x_{St}[assign :: \{x_A\}] \ \varepsilon \ selected\_states \wedge$
$x_S \ [states :: \{x_{St}\}] \ \varepsilon \ next\_locations \wedge$

```
                    x_C[states :: {x_St}] ε next_pieces).
END
```

Figure 50. The view *Successor_Relation*.

Finally, the upward mapping of the view *Init_View* is defined in figure 51:

```
VIEW Init_View
    DEFINITIONS
        CLASS assignments
            ELEMENT_ATT
                    location : {locations};
                    order : {Integer};
                    piece : {pieces};
        END;
        CLASS initial_states
            ELEMENT_ATT
                    assign :: {assignments}
        END;
        CLASS locations
        END;
        CLASS pieces
        END;
        UPWARD MAPPING
            /* 1. The initial states are derived. */
            ∀x_C (NewS(nil,nil)[assign : {NewA(x_C,nil,nil)[piece : x_C, location : nil]}]
                    ε initial_states ←
                        x_C ε components).
            /* 2. All possible assignments are derived (as required by the type-checking) and define
            that pieces are employees and locations are places. */
                    ∀x_C,∀x_S (NewA(x_C,x_S) ε assignments ← x_C ε pieces ∧ x_S ε locations).
                    ∀x_C (NewA(x_C,nil) ε assignments ← x_C ε pieces).
                    ∀x_S (x_S ε locations ← x_S ε place).
                    ∀x_C (x_C ε pieces ← x_C ε employees).
END
```

Figure 51. The view *Init_View*.

5.3 Conclusions From the Exercise

A description of a problem-solving method with KARL can be discussed at three levels. First, KARL provides language primitives to express problem-solving at the knowledge level. Second, KARL has a formal and declarative semantics. Therefore, it allows the precise description of knowledge without requiring the specification of implementational details. Third, KARL is executable, which integrates the benefits of prototyping into the modelling process. Further aspects of KARL are graphical representation of most modelling primitives and means of structuring complex specifications. Some of these aspects are discussed subsequently.

5.3.1 Characterization of Problem-Solving Methods at the Epistemological Level by a Formal Language

[EST92] report that 280 lines of CLIPS code were required to define the application of chronological backtracking for Sisyphus, whereas the board-game method required 40 lines. Our experience with the use of KARL for modelling these problems is consistent with this result. Also, chronological backtracking does not only require an excessive mapping, but also the specification of terminological and inference knowledge at the inference layer (i.e., in stores and elementary inference actions). Thus, a qualitative difference between these two problem-solving methods becomes visible. The chronological-backtracking and the board-game methods need epistemologically different types of knowledge in order to be applicable to the assignment ***task*** in the ***domain*** of the Sisyphus problem:

- The board-game method requires a mapping that transfers the domain-specific knowledge on the generic terminology of the problem-solving method.
- Chronological backtracking requires this mapping and additional knowledge about the task which should be solved with it. It must be defined what a state is (i.e., a set of assignments) and how a successor state is derived from it (i.e., by changing some of the assignments). If chronological backtracking is to be applied for a Sisyphus-like task (i.e., for assignment problems), new definitions and clauses ***at the inference layer*** have to be introduced. Not only domain-specific knowledge must be added but also task-specific knowledge. For example, it must be defined that a state has a specific internal structure. This task-specific knowledge is not needed by the board-game method, because it makes stronger assumptions about the task than chronological backtracking does (cf. [McD88])

Comparing the board-game method with the bottom-up-developed problem-solving method of chapter three, the more complex mapping of the view *Successor_Relation* of the board-game method becomes visible. This view defines this relation in the general term of ***moves*** for the board-game method whereas the Sisyphusmethod makes stronger assumptions about this relation. A successor state is determined by placing one component which is not already placed. Thus, the mapping of the Sisyphusmethod requires less effort to specify it. Apart from that no experience has been gained as to whether the Sisyphusmethod can be reused, whereas the board-game method can in fact be used besides Sisyphus for one-player games like Tower-of-Hanoi (cf. [EST92]). The Sisyphusmethod is very task-specific, and hard-wires domain-specific assumptions (e.g., that a goal state is an assignment where every component has a slot) which is not required for every task and domain.

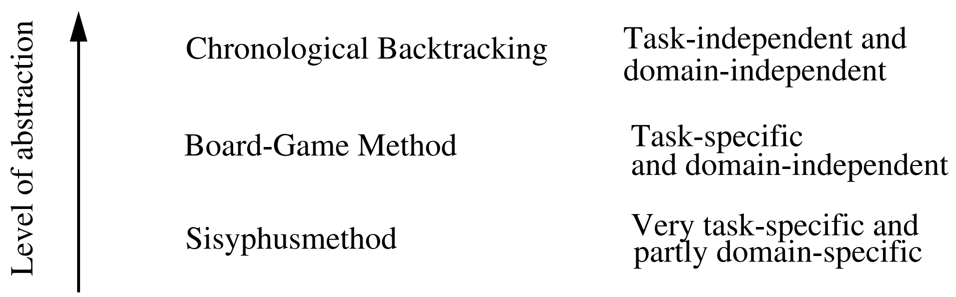


Figure 52. Different levels of generality

In figure 52, these different levels of abstraction are illustrated. The board-game method

restricts chronological backtracking to specific tasks, but is less task-specific than the Sisyphusmethod. The Sisyphusmethod, however, hardwires domain-specific assumptions.
The task-specific formulation of chronological backtracking by the board-game method does not modify the task layer. It is not necessary to specify a task-specific adaptation of the task layer (i.e., the layer which specifies the control flow). The task-specific adaptation is done by terminological knowledge and by refinement of the logical description of inference actions at the inference layer. Is therefore ***task*** layer a misleading name for the layer which specifies the control flow?

5.3.2 Is KARL More Suitable Than a High-Level Programming Language or a Very-High-Level Programming Language?

The formal descriptions of the elementary inference actions of chronological backtracking are mostly trivial and easy to read. Often, one simple clause is sufficient to describe such an inference. The formal descriptions of the elementary inference action of the board-game method are much more complex, which is reflected in the additional task-specific knowledge. These sets of more complex clauses are difficult to read and to understand. However, the clauses define the exact meaning of these actions, and doing this in the case of a nontrivial action is a difficult task. The goal of KARL is to simplify these precise descriptions of the elementary inference actions as much as possible by abstracting from all implementational details. The knowledge engineer describes declaratively ***what*** an elementary inference action does without referring to ***how*** it is done, that is, how it is implemented by a computer program. If a programming language like "C", for example, were to be used one would additionally be concerned with aspects related to the fact that a computer is the agent of problem-solving. Consider a simple KARL clause like:

$\forall x\ (P(x) \leftarrow Q(x)).$

A corresponding C-code is shown in figure 53.

```
struct ELEMENT
{
    int item;
    ...
    struct ELEMENT *next;
}
/* The function copies the list l2 and adds this copy at the end of l1. It returns l1. */
struct ELEMENT *append(l1,l2)
struct ELEMENT *l1;
struct ELEMENT *l2;

{
    struct ELEMENT *e;
    struct ELEMENT *l;

    /* Search for the end of l1. */
    l=l1;
    while (l->next!=NULL)
        l=l->next;
    /* Copy the elments of l2 and add it at the end of l1. */
```

```
    while (l2!=NULL)
    {
        /* Allocate storage. */
        e=(struct ELEMENT *)malloc(sizeof(struct ELEMENT));
        ....
        /* Copy element. */
        e->item=l2->item;
        ...
        e->next=NULL;
        /* Add element to l1. */
        l->next=e;
        l=l->next;
        l2=l2->next;
    }
    return l1;
}
```

Figure 53. C-code

Comparing KARL with a very-high-level programming language like PROLOG the main distinctions are:

- PROLOG includes several features that does not have a declarative semantics like assert, delete, fail, or “!”.
- PROLOG does not provide means that allow the explicit representation of control flow.
- The ordering of clauses and premises in clauses can be used to determine implicitly the control flow of a problem-solving process. This problem is well-known from production-rule formalisms where the ordering of premises are used to represent control.
- Negation as failure as applied by PROLOG is incorrect in regard to logical negation and incomplete in regard to the perfect model semantics.
- PROLOG does not provide appropriate language primitives to express static knowledge. Only constants and predicates are given. KARL provides object-orientation, functional and set-valued attributes, well-typing, is-a and is-element-of relationship, multiple attribute inheritance, etc.

5.3.3 A Concluding Remark

It is the ***dual*** character of KARL, i.e. its formal character and its knowledge-level primitives, which gave me new insight in the task-specific refinement of problem-solving methods. I had quite a different conceptual view on how to refine a general method to a task-specific strong method. But when applying KARL, which contains the primitives that explicitly distinguish between epistemologically different kinds of knowledge, I came up with a very different conceptualization of this task. It was the “knowledge-level character” of KARL which made this possible. In addition, it was the formal character, i.e. KARL´s precise semantics, which made it a necessary result. When using KARL, he forced me to model the discussed methods in a precise manner. Neither only a formal language which would end up in an amorphous set of clauses, nor only a informal knowledge-level language would have this strong guiding feature. It is the combination of both which allows its user to detect new (and hopefully correct) ways of conceptualization of a phenomenon. You just use this language and watch what conceptualization it is detecting for you. Only a formal language with the appropriate epistemological primitives can perform this job for you.

Chapter 6

A Comparison of KARL with Other Approaches

In this chapter, KARL is compared with other languages. First, a comparison with other KADS-oriented knowledge specification languages is given. Then KARL is discussed in the context of software engineering and information systems development.

6.1 A Comparison with Other KADS-Oriented Specification Languages[1]

During the few last years, a lot of work has been accomplished to overcome the disadvantages of knowledge level specifications with an informal semantics only. The increasing number of specification languages which allow a formal and executable specification of the modelled expertise at the knowledge level try to overcome the shortcomings of life cycle-oriented development processes. Informally specified documents like the KADS model of expertise receive a defined semantics. Three different kinds of languages can be identified (cf. section 1.1):

- Languages which can be used to ***operationalize*** models of expertise. These languages add the advantages of prototyping, i.e. the evaluation by a running program to the modelling process.
- Languages which can be used to ***formalize*** models of expertise. These languages allow a precise formulation of a model of expertise and make checks for consistency, redundancy, and correctness possible by verification techniques.

1. I would like to thank Frank van Harmelen for his cooperation. The comparison of KARL with other knowledge specification languages was done in cooperation with him (cf. [FeH93]).

- Languages which attempt ***both***. These languages aim integrating the advantages of formal and operational languages.

Until now, the following languages have been discussed in the research literature:

- FORKADS (cf. [Wet90], [WeS91], [Wet92]),
- KARL (cf. [FAL91], [AFS93]),
- $K_{BS}SF$ (cf. [JoS92], [VJS93])
- $(ML)^2$ (cf. [AHS93], [HaB92]),
- Model-K (cf. [KVS91], [KaV92], [KaV93]),
- Momo (cf. [LKV92], [VVW93], [WVL92])
- OMOS (cf. [Lin92a], [Lin92b]),
- and QIL (cf. [ARS92], [AKS93]).

In addition, [Möl92] proposes KL-ONE as an operationalization and formalization language. In a more elaborated way, this idea is realized by the language MODEL/KADS (cf. [Bar93]). [Gei92] defines an operational language in the domain of decision support systems which can also be viewed as an operational knowledge specification language because it is based on the KADS models of expertise. A further knowledge specification language, which is not discussed in the KADS-context, is DESIRE (cf. [KoT90], [LPT93]).

In the following, KARL is compared with three of these languages. It is compared with MODEL-K, which aims to be an operational language, with FORKADS, which aims to be an operational and formal language like KARL, and with $(ML)^2$, which aims to be a formal language. To keep the thesis self-contained, a short sketch of these three languages is given. A thorough comparison with all of these operational or formal knowledge specification languages is beyond the scope of this thesis (cf. [FeS92] and [FeH93] for further comparisons). Before the description of the languages MODEL-K, FORKADS, and $(ML)^2$ and the comparison with these languages is given, different levels and criteria for this description and comparison are discussed.

6.1.1 How to Compare the Different Languages

For analysing the various languages, comparison criteria are required. The definition of these criteria is important since it will clearly influence the result of the present investigation. What are the relevant criteria to investigate languages used to specify a model of expertise? Criteria which can be applied to compare programming languages seem hardly suitable, since such languages have entirely different purposes.[1] Programming languages should allow an effective and efficient realization of the modelled expertise at the symbol level, and should therefore deal with other aspects than the specification languages studied here.[2]

---

1. See [AnK92] for a comparison of Modula with C.

2. Compare [SAW89] for the relationship of adequate knowledge level modelling of expertise and efficient computational solutions.

As stated above, the languages which are compared here allow a ***formal*** and ***executable*** specification of models of expertise ***at the knowledge level***. Therefore, three different kinds of questions seem relevant in the comparison of these languages (cf. [Bra85]):

- **Epistemological level:** Does the language provide ***epistemologically*** adequate modelling primitives like the four-layered KADS model of expertise to allow a specification at the knowledge level?
- **Formal level**: Is the language meant as a tool for ***formalising*** a model of expertise? This is a question concerning the semantics of the provided modelling primitives.
- **Operational level:** Is the language meant as a tool for ***operationalizing*** the conceptual model? This is a question concerning the adequacy of a language to support executability and interpretation.

The following subsections discuss a list of comparison criteria which are organized according to these three levels.

6.1.1.1 Criteria at the Epistemological Level

In [BrW89], [WSB92], or [Cla92a] several language primitives are defined which can be used to express different kinds of knowledge and their relationship. The languages should contain these or adequate primitives to allow the formulation of the model of expertise at the appropriate level. At this level, which we will call the ***epistemological level***, the modelling power of a language is discussed. It is obvious that the criteria at this level are highly dependent on the kind of conceptual models at which the languages are aimed. As mentioned before, the comparison here is concentrated on languages for KADS models to make a meaningful comparison possible. The following questions are of interest here:

- Which language primitives are provided to express ***domain*** knowledge?
- Which language primitives are provided to express ***inference*** knowledge?
- Which language primitives are provided to express ***task*** knowledge, i.e. knowledge about the control flow of a problem-solving process?
- Which language primitives are provided to express ***strategic*** knowledge?
- How are the different kinds of knowledge ***connected***, i.e. how is the idea of a reusable and therefore generic inference knowledge integrated?
- As specifications of large systems become unreadable it is necessary to ask for every layer whether there are means of structuring the modelled expertise.

6.1.1.2 Criteria at the Formal Level

The languages should have a formal semantics to allow a unequivocal and well-defined formulation of the model of expertise. Therefore, relevant questions at this level, called further on the ***formal level***, are:

- What is the ***purpose*** of the language, i.e. is it intended as a tool for formalizing a model of expertise or is it only an operationalization language?
- What are the ***semantics*** of the available modelling primitives? Does the language have a formal semantics or is the semantics only defined informally or by an interpreter? Which kind of formal semantics does a language have? Is it only a declarative semantics or does it also have an operational one?[1] If so, how are these two related?
- How are ***dynamics*** and change represented in a declarative framework?
- What is the ***expressive*** power of a language?

6.1.1.3 Criteria at the Operational Level

The language should be executable in order to allow the evaluation of the model of expertise by prototyping. Therefore, questions at this level, further on referred to as the ***operational level***, are:

- What is the ***purpose*** of the language, i.e. is it intended as a tool for operationalizing a model of expertise or is it only a formalization language?
- What is its ***expressive power***? It is easy to provide a language with a high expressive power, a natural language, for example. It will, however, become very difficult to make it run or even walk on a computer.
- Are there any comments about ***efficiency***?
- If the language is intended for operationalization and formalization, how are these two aspects related? Does the implemented ***semantics*** differ from the operational or declarative semantics?

6.1.2 Model-K

MODEL-K (cf. [KVS91], [KaV92], [KaV93]) is a language for operationalizing KADS models of expertise. It has been developed at the German National Research Centre for Computer Science, GMD in Bonn. It is implemented in BABYLON (cf. [CPV89]).

The main point in MODEL-K is the use of an existing AI shell, i.e. BABYLON, to operationalize KADS model of expertise. This is done by introducing predefined modelling primitives in BABYLON. An operational model can be built using terms of a knowledge level

---

1. A declarative semantics of a language defines the meaning of its primitives in a second "well-known" language. This can be done in a non-constructive way, i.e. there need not exist any possible algorithm which can compute this semantics. An operational semantics defines the semantics of a language in a constructive way, i.e. it defines an algorithm which can compute it. This should not be confused with the distinction of formalization and operationalization languages. An operational semantics is a kind of link between a declarative semantics and a running interpreter. But often, the declarative semantics differs from the operational one ***and the operational one differs from the implemented semantics***. An operational semantics defines an algorithm only in principle, similar to a constructive proof in mathematics. An executable or implemented semantics must be run on a real computer. The three different kinds of semantics are answers to three different kinds of questions: Is there a formal semantics, a model-theoretic one, for example, is there a proof procedure for it, and can it be implemented effectively and efficiently?

specification. MODEL-K allows the evaluation of the model of expertise by prototyping. Additionally, MODEL-K is considered as a language for the implementation of a final expert system. An incremental development of the expert system and a system which reflects the conceptual structure of the knowledge level description of the expertise become possible. The conceptual structure can become a powerful tool for improving understandability and maintenance. However, as discussed above, essential parts of the operationalized expertise must be described by programming it. MODEL-K consists of two sub-languages. One of these is neither operational nor has it a formal semantics. This language can be used to describe the conceptual structure of the expertise. In addition, a second sub-language can be used to implement the meaning of the specified model using BABYLON. Therefore a specified expertise in MODEL-K is highly related to the symbol level.

Another important feature of MODEL-K is its possibility of modelling reflective problem solving. This can be considered as a step in the direction of a strategic layer.

6.1.2.1 The Epistemological Level

MODEL-K provides languages for all four levels of a model of expertise, i.e. for the domain layer, the inference layer, the task layer, and the strategic layer.

MODEL-K consists of two sub-languages (cf. figure 54). MODEL-K provides a ***specification language*** which allows an informal description of the expertise. In addition, MODEL-K provides an ***operationalization language*** which can be used to operationalize the terms defined by the specification language.

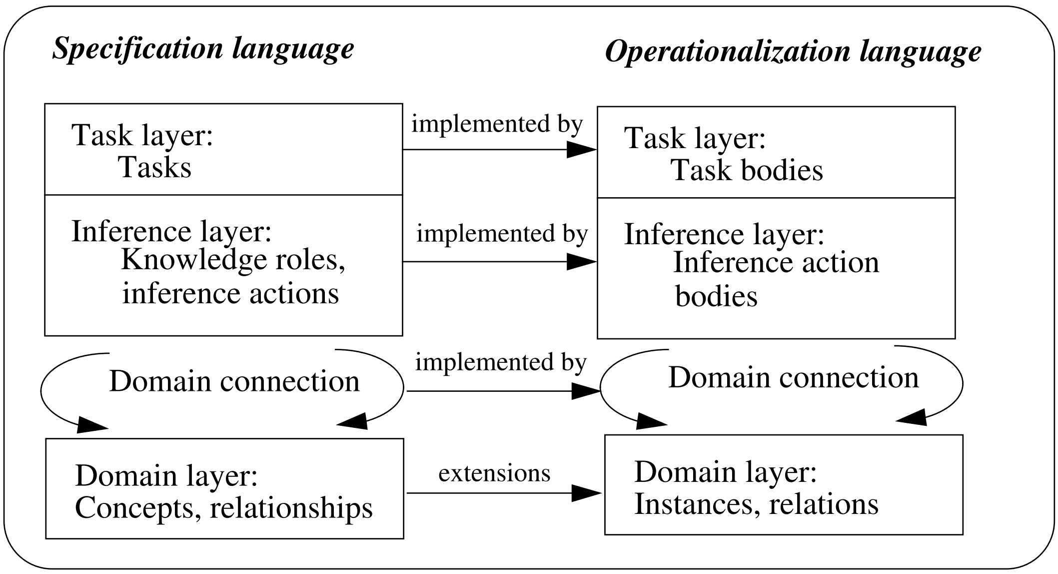


Figure 54. The two sub-languages of MODEL-K

The Domain-Layer Primitives

The specification language can be used to model terminological knowledge by concept hierarchies and relationships between concepts. Frames of BABYLON are used to model concepts and relationships.

Assertional knowledge can be modelled with the operationalization language by instances of concepts and tuples which are elements of relations. These relations can be described extensionally or intensionally.

The Inference-Layer Primitives

The specification language can be used to define inference actions, i.e. their names, and their input and output knowledge roles. These definitions can be compared with interface definitions of modules. The external behavior of an inference action is defined as well as the whole inference structure, but it is not specified how a inference action draws an inference: they are considered as black boxes. The description of the problem-solving method is therefore neither executable nor has it a formal semantics. The inference structure is implicitly defined by the connection of inference actions and knowledge roles and cannot be labelled explicitly. Therefore, MODEL-K allows no hierarchical structuring and refinement of inference structures.

The operationalization language allows the operational refinement of inference actions by adding an inference-action body. This body implements the behavior of inference actions using BABYLON code. Thus, the operational description of the problem-solving method is a result of a programming step.

BABYLON provides powerful mechanisms for implementing inference-action bodies, i.e. Lisp, PROLOG, constraints, etc.

The Task-Layer Primitives

The specification language can be used to informally describe tasks and their hierarchical refinement. The operationalization language adds task bodies allowing the specification of control knowledge. It provides a procedural language consisting of sequence, branch, and loops.

As the knowledge roles are modelled as containers, two update commands are given. A knowledge role can be filled with domain knowledge, and the contents of a knowledge role can be written back to the domain layer.

The Strategic-Layer Primitives

MODEL-K supports the modelling of reflective problem-solving. A model of expertise containing reflective modules, consists of three different levels, cf. figure 55:

- An object system is modelled which consists of domain layer, inference layer, and task layer.
- Several reflective modules are specified, which again consist of domain layer, inference layer, and task layer. The difference is the fact that their domain layers contain the model of the object system.
- A scheduler defines the control flow between the reflective modules.

An example for reflective problem-solving is given in [KaV93]. An object system should assign employees to places in consideration of several constraints (i.e., the Sisyphusproblem). The reflective problem-solver then deals with the following questions:

- One reflective module enables the user to restrict the maximal number of seconds to be spent and the maximum number of solutions to be found by the object-system (cf. [KaV93]).
- If the input problem is too complex it might be useful to decompose this problem into a sequence of increasingly complex sub-problems and solve those stepwise (cf. [KaV93]).

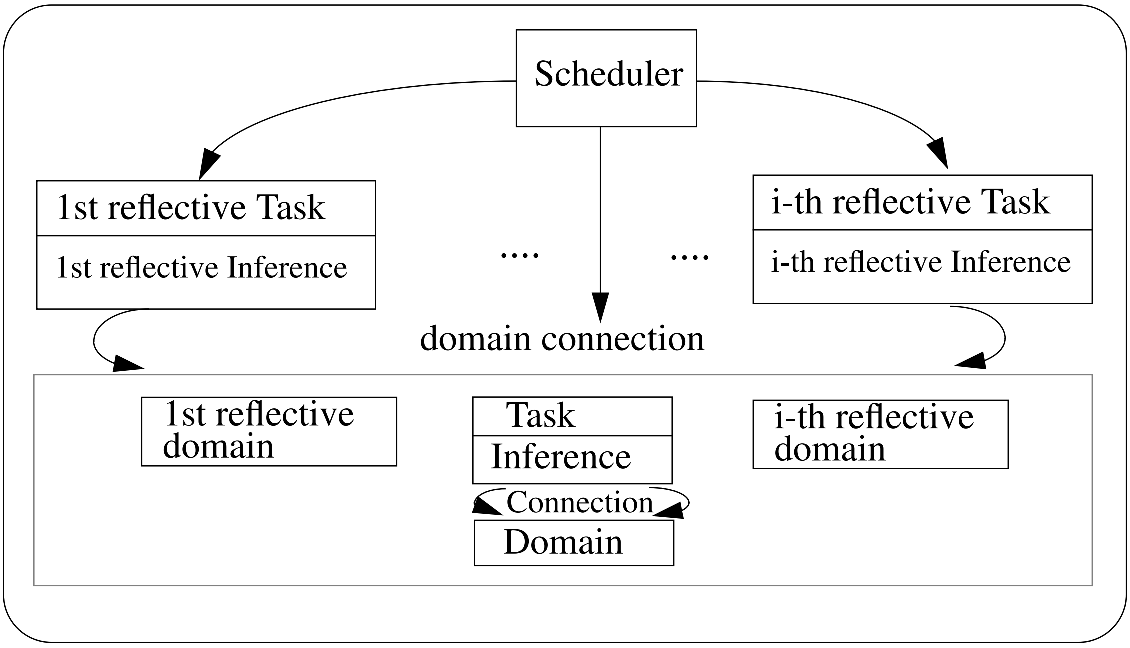


Figure 55. Modelling reflective problem-solving with MODEL-K

The modelling primitives for reflective problem-solving do not increase the expressive power of the modelled problem-solver. However, they allow modelling reflective problem-solving at the conceptual level and therefore support understandability and reusability.

The Primitives Connecting the Domain Layer and Inference Layer

The specification language introduces names to express the mapping between domain layer and inference layer. The semantics of this mapping is defined by the operationalization language. Knowledge roles and inference actions are connected to domain expressions by means of the BABYLON code.

The Primitives Connecting the Inference Layer and Task Layer

BABYLON code is used to implement these connections in the task bodies.

6.1.2.2 The Formal Level

MODEL-K is not discussed as a formal specification language but as an operationalization language only. The specification language only has an informal semantics. The knowledge model receives an executable semantics if the inference action and task bodies are implemented in BABYLON. The executable semantics is defined by the BABYLON interpreter.

6.1.2.3 The Operational Level

MODEL-K provides the expressive power of programming languages like Lisp or PROLOG, i.e. it is Turing-complete. Special tools for supporting knowledge formalization, evaluation, assessment, or error correction are not discussed. MODEL-K contains the environment of BABYLON.

6.1.2.4 Applications

MODEL-K has been applied to the following problems:

- ***Assignment***: The Sisyphusproblem (cf. [VKS91], [DKV92]) and the assignment of airplanes to gates (cf. [VoK93]).
- In addition, ***reflective modules*** have been developed in the context of the Sisyphusproblem (cf. [KaV93]): a case-based reasoner, a scheduler for the management of computing time, and several modules which transform or simplify problem descriptions (cf. [BBC91]).

6.1.2.5 Discussion

MODEL-K is not a formal specification language for the model of expertise because it has no declarative semantics. The specification language only has an informal semantics. It receives an implemented semantics by programming the inference action and task bodies with the help of the operationalization language, i.e. BABYLON. It does not allow to declaratively describe what an inference action is doing but it is necessary to implement how it is done. The prototype which can be used to evaluate the specified expertise is the result of an implementation step, and a high amount of control knowledge can be hidden in the implemented bodies. Therefore, MODEL-K is not really a specification language. Also, BABYLON is a very heterogeneous language providing Lisp, PROLOG, Frames, etc. This reduces the understandability and explicitness of the modelled expertise.

As the running prototype is the result of an implementation step, MODEL-K violates the separation of the knowledge level and the symbol level. Apart from that, MODEL-K is a powerful and expressive language for implementing running expert systems while preserving the conceptual structure of the model of expertise. MODEL-K adds symbol-level control to the specification and therefore allows more efficient specifications than the declarative descriptions of inference actions in KARL. KARL and MODEL-K have the same expressive power.

The most significant differences between MODEL-K and KARL at the epistemological level are:

- MODEL-K does not contain hierarchical refinement at the inference layer.
- MODEL-K does not distinguish between data and knowledge, neither at the domain layer nor at the inference layer.
- MODEL-K has a strategic layer.

6.1.3 FORKADS

FORKADS has been developed at the IBM Germany Scientific Centre in Heidelberg. It was one of the first published approaches to formal KADS models (cf. [Wet90], [WeS91], [Wet92]). Contrary to MODEL-K, FORKADS aims not only at operationalizing KADS models, but also aims at giving solid formal foundations to KADS models. For this purpose, the main foundation of KADS is a first-order logical language which is extended with notions of concept heterarchies and procedural attachment. FORKADS is rooted in the LILOG language (cf. [HeR91]).

6.1.3.1 The Epistemological Level

FORKADS is based on order-sorted logic with certain extensions.

The Domain-Layer Primitives

In FORKADS, concepts are represented as sorts, and instances of a concept as constants of those sort. Sorts (concepts) can have a lattice-like subsumption relation for modelling subconcepts. So-called features and roles of concepts are represented by one-place functions and two-place relations over sorts. Other relations are modelled as n-place predicates.

The features mentioned above represent the general “domain model” of a FORKADS domain layer. Properties of individual instances of sorts, or the truth conditions on relations, roles, and features are specified by a set of first-order axioms.

The Inference-Layer Primitives

Two types of inferences are distinguished at the inference layer: terminological and referential inferences. The terminological inferences deal with properties represented in the sort-structure. Examples of such terminological inferences are

- *identify(VarorConst, S)* which makes *VarorConst* of sort *S*;
- *allocate(C,S)* which creates a new constant *C* of sort *S*; and
- *dissect(S,R,V)* which binds variable *V* to the set of direct subsorts of sort *S* which are in relation *R* with *S*.

The referential inferences infer properties of specific individuals. The two main primitive referential inferences are:

- *holds(Goal)* which tries to prove *Goal*, and
- *solves(V,Goal)* which tries to establish a binding for variable *V* by proving or executing *Goal* (provided that *Goal* is ground except for *V*); and
- *known(V)* tests if a value has been computed for variable *V.*

All these are primitive inference actions and are used to build non-primitive inferences. Each non-primitive inference is given a so-called “inference-head”, which consists of the name of the inference and its parameters which can be used for calling the inference.

The Task-Layer Primitives

As with inferences, tasks and subtasks are also expressed by means of a task-head and a task-body. The task-head contains name and parameters and can be used to call the (sub)task.

Bodies of tasks and subtasks are a sequence of statements which consist of calls to other (sub)tasks or inferences (which are considered as primitive tasks). Furthermore, task bodies can contain the following constructs:

- a loop statement *USEALLSORT V in S; <BODY>* which iterates *Body* over all elements of sort *S*, or
- if-then-else statements.

Two other types of statements are allowed in task bodies, and these are applied to affect the set of axioms used for proving inferences:

- *USE-AXIOMS(T)* allows future use of subtheory *T* during inference, and
- *ASSERT(L)* adds literal *L* to the knowledge-base.

The Strategic-Layer Primitives

FORKADS does not deal with the strategic layer.

The Primitives Connecting Domain Layer and Inference Layer

FORKADS distinguishes between a specific domain model and a generic “stereotype”. Both are sort structures with their associated relations, but a domain model describes a specific application domain, while a stereotype describes a generic structure. Such stereotypes are used to define domain-independent inference structures and task layers (interpretation models). The connection between a specific domain model and the generic sort model of a stereotype is made by statements of the form *role_of(.,.)*, which link sorts and relations of the domain model to their counterparts in the sort model of the stereotype.

The Primitives Connecting Inference Layer and Task Layer

Since inferences are considered as primitive tasks, and no further separation is made between task layer and inference layer, no separate mechanism is required to connect the two layers.

6.1.3.2 The Formal Level

The domain layer of FORKADS consists of an order-sorted, first-order predicate calculus. It is the only layer in FORKADS with a declarative semantics. Neither the inference layer nor the task layer are given a declarative semantics. These two layers are only given a operational semantics in terms of an abstract interpreter. However, the properties of these semantics are hard to assess since their construction is along rather unconventional lines. The expressive power of the used formalism is not determined by the author of the language.

6.1.3.3 The Operational Level

[BüW92] sketch an implementation of FORKADS consisting of a graphical modelling component GCONMOD and a code generator LCONMOD. LCONMOD generates PROLOG code which is used to execute FORKADS inferences. It can use procedurally implemented predicates in C and uses the inference machine of the knowledge representation language L_{LILOG} (cf. [HeR91]). The authors do not discuss efficiency of the implemented interpreter for FORKADS. The graphical tool GCONMOD allows editing and browsing of FORKADS structures either in terms of formulae, or in terms of a graphical representation of the dependencies in a FORKADS model.

As FORKADS does not have a declarative semantics, there is no need to determine the relationship of the declarative and operational semantics.

6.1.3.4 Application

FORKADS has been used in a commercial banking application project (cf. [Wet92]).

6.1.3.5 Discussion

FORKADS closely follows the usual KADS interpretation of domain layers in terms of instances and concepts, representing them by well-understood notions such as constants and sorts. KARL provides a richer variety of language primitives.

The task layer of FORKADS uses nonstandard constructs (e.g. the *USEALLSORT* construction), but otherwise contains common notions such as sequence or alternative. FORKADS' most unusual features appear in the inference layer. The referential inference constructs like *known* and *solves* and terminological ones like *allocate*, as well as the possibility for procedural attachment require a nonstandard semantics which is considerably different from a standard semantical account.

FORKADS lacks a formal (i.e., declarative) semantics, which restricts is usefulness as a formalization language.

Other than KARL, FORKADS provides a set of predefined inference actions.

6.1.4 (ML)^2

(ML)^2 (cf. [AHS93], [BaA92b], [HaB92]) is a language for formalizing KADS models of expertise.[1] It was developed in the course of the ESPRIT Projects 3178 "REFLECT" and 5248 "KADS-II" and a bilateral research project of the Netherlands Energy Research Foundation ECN and the University of Amsterdam. Although a subset of (ML)^2 can be operationalized to allow explorative prototyping (cf. [THR91]) it is mainly introduced as a formalization language allowing the unequivocal formulation of models of expertise. The significant feature of (ML)^2 is its uniform application of logic to the different layers of expertise, especially to the representation of the task layer. It provides several embedded types of logic to declaratively describe a model of expertise. Modularized order-sorted first-order logic is used to describe a domain layer. The inference layer is modelled by meta-logic and the task layer is modelled by quantified-dynamic logic.

6.1.4.1 The Epistemological Level

(ML)^2 provides language primitives for the first three levels of a model of expertise, i.e. for the domain layer, the inference layer, and the task layer.

The Domain-Layer Primitives

The sublanguage of (ML)^2 used to model a domain layer is order-sorted first-order logic extended by modularization. Instances are modelled by constants, and sorts can be applied to model classes of such constants. Sorts can be arranged in an is-a hierarchy. Relationships between concepts are modelled by predicates. Attributes of concepts are modelled by functions. Arbitrary first-order theories can be used to specify the defined relationships. The specification of a domain layer can be divided into several modules. Such a module or theory defines a signature (i.e. sorts, constants, functions, and predicates) and axioms (i.e. logical formulae). These modules or subtheories can be combined by a union operator.

The Inference-Layer Primitives

(ML)^2 provides primitive inference actions, knowledge roles, and the linkage of both as language primitives at the inference layer. A primitive inference action (pia) can have several input-knowledge roles, but has exactly one output knowledge role. Every inference action and every knowledge role is described by a theory similar to domain layer theories. The theory which describes a knowledge role defines the mapping of domain layer terms in terms used at the inference layer. A theory describing a primitive inference action imports all theories of its input-knowledge roles and output-knowledge roles. Its axioms describe the inference drawn by the inference action using the terms of the input-knowledge roles and output-knowledge

1. An earlier version of this subsection was published in [FeS92].

roles. Figure 56 illustrates an inference layer modelled in $(ML)^2$.

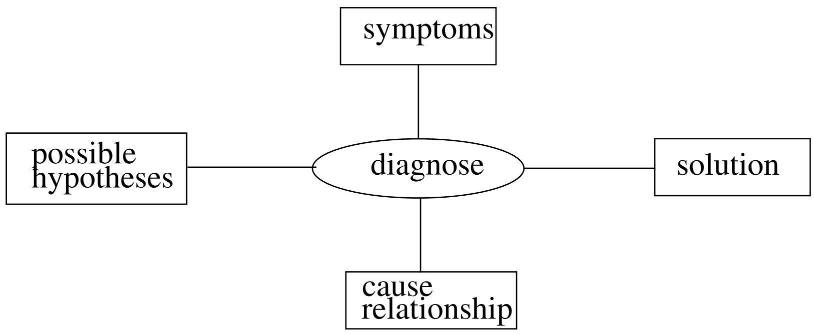


Figure 56. A brief inference structure in $(ML)^2$

The primitive inference action (pia) *diagnose* diagnoses a *solution* using *symptoms*, *possible hypotheses*, and a *cause relationship* as input. Note that the *cause relationship* is modelled as a view in KARL whereas in $(ML)^2$ it becomes an input-knowledge role. This pia *diagnose* is modelled by a predicate:

$pia_{diagnose}$(symptoms(X), possible_hypotheses(Y), cause_relationship(symptoms(X), possible_hypotheses(Y)), solution(Y))

For a description of an inference it is possible to use a set of clauses with the predicate given above or intermediate predicates as their conclusions and the input-knowledge roles or intermediate predicates as premises. The whole description of the pia is:

theory *diagnose*
  use symptoms, possible_hypotheses, cause_relationships
  signature
    predicates $pia_{diagnose}$
  variables X, Y
  axioms
    $pia_{diagnose}$(
      symptoms(X),
      possible_hypotheses(Y),
      cause_relationship(symptoms(X), possible_hypotheses(Y)),
      solution(Y)
) $\leftarrow$
    $input_{symptoms}$(symptoms(X)) $\wedge$
    $input_{possible\_hypotheses}$(possible_hypotheses(Y)) $\wedge$
    $input_{cause\_relationship}$(cause_relationship(symptoms(X),
    possible_hypotheses(Y))

Hierarchical refinement of inference structures is not discussed by the authors of $(ML)^2$.

The Task-Layer Primitives

Quantified-dynamic logic is used to specify dynamic control at the task layer. Every predicate specifying a pia at the inference layer is considered as an elementary program statement and the knowledge roles are used as input parameter and output parameter of such programs. For every such elementary program a history variable V_{pia} is defined which stores the input-output pairs for every execution step. Four types of queries are available for such programs described here for the pia "diagnose":

- ***has-solution***-$pia_{diagnose}(I_1, I_2, I_3, O)$: The program checks whether the predicate $pia_{diagnose}$ is evaluated to be true by applying the axioms of its theory. I_i and O can be variables or instantiations for which the predicate is proved.
- ***old-solution***-$pia_{diagnose}(I_1, I_2, I_3, O)$: The program asks whether (I_1, I_2, I_3, O) matches with an input-output pair stored in $V_{diagnose}$.
- ***more-solution***-$pia_{diagnose}(I_1, I_2, I_3, O)$: The program asks whether a solution (I_1, I_2, I_3, O) exists which does not match with an input-output pair stored in $V_{diagnose}$.
- ***give-solution***-$pia_{diagnose}(I_1, I_2, I_3, O)$: This program first checks if a previously uncomputed solution exists, and if so, stores it in the corresponding Variable $V_{diagnose}$. This language construct can be used to define a single step of a program. These single steps can be combined by sequence, branch, and loop and arranged to form subtasks..

In KARL, for example, the stores work as containers holding the current results of the inference actions. In $(ML)^2$ this task is performed by the history variables V_{pia}. In addition, these variables do not store one single state of the problem-solving process, but ***all*** states of the problem-solving process.

The Primitives Connecting the Domain Layer with Inference Layer and Task Layer

The inference layer and task layer are modelled as a meta-language of the domain layer. Object-language and meta-language are connected by a naming relation and reflection rules.

At the inference layer, a theory and a lift operation are defined for every knowledge role which is connected to the domain layer. The lift operator defines a naming relation by mapping terms of the domain layer to variable-free expressions at the inference layer. This is similar to the view definition of KARL but has more expressive power. Other than in KARL, not only instances or classes but arbitrary domain terms, predicates or clauses can be mapped to terms at the meta-layer.

At the task layer, *downward* and *upward* connections between domain layer and inference/task layer can be defined. This is done by applying three reflection rules:

$$\frac{\phi \in O}{\vdash_M \mathit{ask}_{\in}(O, \lceil \phi \rceil)} \quad (\mathrm{ask}_{\in}\text{: upward}) \qquad \frac{\phi \in O}{\vdash_M \mathit{ask}_{\vdash}(O, \lceil \phi \rceil)} \quad (\mathrm{ask}_{\vdash}\text{: upward})$$

$$\frac{\vdash_M \mathit{tell}(O, \lceil \phi \rceil)}{\phi \in O} \quad (\text{tell: downward})$$

M = meta-theory
O = object theory

$ask_{\in}$ is true for ϕ if ϕ is an axiom of the domain layer. $ask_{|-}$ is true for ϕ if ϕ is derivable at the domain layer. *tell* stores ϕ as an axiom at the domain layer after ϕ has been derived at the inference layer.

The Primitives Connecting the Inference Layer and Task Layer

The predicates which characterize an inference action return a boolean value and a variable binding. As the task layer is not modelled as a meta-layer to the inference layer but as an extension of it, no reflection rules are necessary.

6.1.4.2 The Formal Level

$(ML)^2$ provides three kinds of logical languages to describe models of expertise:

- ***Order-sorted first-order logic*** extended by modularization at the domain layer. This language is semi-decidable, i.e. it does not have a complete proof-theory.
- ***Meta-logic*** at the inference layer.
- ***Quantified-dynamic logic*** at the task layer. It is a modal extension of first-order logic. A state is characterized by the values of all variables and the true values of elementary formulae. History variables are defined which store all states during problem solving.

$(ML)^2$ uses the model-theoretical semantics of first-order logic as semantics for static knowledge and the Kripke semantics of dynamic logic as semantics for dynamic knowledge. The expressive power of finite axiom sets in $(ML)^2$ exceeds Turing-completeness.

6.1.4.3 The Operational Level

In [BaA92a] TheME is described which supports the formalization of models of expertise using $(ML)^2$. It allows a graphical representation of $(ML)^2$ expressions and checks for syntactical correctness but it does not help to check the semantics of the modelled expertise.

In [THR91] a prototype interpreter $Si(ML)^2$ for a subset of $(ML)^2$ is discussed. The main restriction concerns the domain layer. It is restricted to Horn logic, and negation is interpreted as negation as failure.

6.1.4.4 Applications

$(ML)^2$ has been applied in a dozen formalizations of models of expertise (cf. [HaB92], [BaA92b]), e.g., for making predictions about physical systems by means of qualitative reasoning (cf. [BRW90]). The problem-solving methods ***cover and differentiate*** (cf. [SWA92]) and ***heuristic classification*** (cf. [SWA92]) are formalized in $(ML)^2$. A scheduling task with simple search heuristics has been specified in [BHA93].

6.1.4.5 Discussion

$(ML)^2$ applies different types of logic for formalizing a model of expertise. First-order logic, meta-logic and quantified-dynamic logic are instruments for modelling the different layers. $(ML)^2$ was the only language which provided a logical description for the control knowledge at the task layer (this solution has later been adopted by KARL). The application of meta-logic for the inference layer comes close in spirit to the role of the inference layer.

Apart from these advantages, there are still some problems with $(ML)^2$. On the one hand, $(ML)^2$ provides a powerful language for modelling domain knowledge. On the other hand, this language is only semi-decidable. It is no surprise that the operationalized subset of $(ML)^2$ is restricted to Horn logic and has a different semantics (i.e., negation as failure).

The frame-based representation formalism of KARL provides a greater variety of different modelling primitives to model the domain knowledge. On the one side, a uniform representation formalism like predicate logic is easier to learn than richer representation formalisms, because it provides only a small number of different modelling primitives (i.e., constants and predicates). On the other side, a uniform representation formalism makes it necessary to express epistemologically different types of knowledge by the same modelling primitives. Also, $(ML)^2$ does not distinguish between data and knowledge.

$(ML)^2$ restricts the number of output-knowledge roles to one single role, which forces the modeller to introduce additional inference actions. For example, if a set is to be split into two parts this fact must be modelled by two inference actions. Hierarchical refinement, which helps to structure complex inference structures, is not discussed.

The inference layer is a real meta-layer in $(ML)^2$. Therefore, it can in principle be used to model inferences about arbitrary domain terms. In KARL, domain layer and inference layer are modelled by the same stratified Horn logic, i.e. by L-KARL rules. The first version of KARL as published in [FAL91] was not able to reason about classes, i.e. predicates. In the case of a classification problem the final classes had to be modelled as constants to allow their binding on variables as a result of a query. Modelling individuals and terminal classes by constants and super-classes by predicates is, however, a very artificial way. It does not reflect a modelling decision but is an artefact of the modelling language. The same problem would arise in order-sorted logical languages. The sorts define hierarchical domain restriction for the variables but it is not possible to instantiate a variable directly by a sort, i.e. the sorts hierarchy cannot by used to express a domain hierarchy which is to be browsed. In F-logic and in the same manner in L-KARL this goal is reached by semantically handling (i.e., interpreting) classes by individuals. Thus KARL defines a intermediate position between predicate logic on the one hand and meta-logic on the other hand. KARL cannot reason about arbitrary terms of an object layer, but the following clause shows how all super-classes of a class can be bound on a variable:

$$\forall x\ (\text{superclass}(\text{class} : c, \text{superclass} : x) \leftarrow c \leq x).$$

In $(ML)^2$ the whole trace of the problem-solving process is stored in the history variables V_{pia} corresponding to every pia. For complex reasoning-tasks this would probably cause problems in the operationalization. This point is an example of how $(ML)^2$ is more aimed at formalization than at operationalization. This difference appears also when comparing the differing expressive power of the conditions at the task layer in KARL and $(ML)^2$. Whereas KARL allows only very restricted possibilities to express conditions, $(ML)^2$ provides arbitrary formulae of quantified-dynamic logic.

In $(ML)^2$ it is not possible to ***delete*** contents of knowledge roles, i.e. history variables. The ***old-solution***-pia$_{diagnose}$(I_1, I_2, I_3, O), for example, asks for all already computed solutions. This restriction is related to the tuple-oriented evaluation strategy of $(ML)^2$. The previously solutions of an inference action have to be kept in mind to prevent $(ML)^2$ from repeatedly regarding the same tuple as its solution.

In general, $(ML)^2$ is designed mainly as a formalization language allowing the declarative description of models of expertise. But experience shows that even for experts in logical formalisms it becomes difficult to really understand the meaning of formally specified knowledge, especially if the model becomes very large. The operationalization of merely a subset of the language cannot totally overcome this shortcoming, because of differences in the semantics of the formal and the operational language.

6.1.5 Conclusion

The discussed languages just discussed are rather complementary than contradictory. In the first step a language like $(ML)^2$ can be used to formalize a model of expertise. For the translation of an informally specified expertise into a formal representation it is helpful to have as much expressive power as possible. A language like KARL can be used in a second step to add the flavour of executability to the specified expertise. The price for this is the restriction of the expressive power of the language. A language like MODEL-K can be applied to add the aspect of efficiency to the specified expertise by implementing the bodies of the knowledge sources. This supports the change to design and implementation without destroying the conceptual structure of the modelled expertise. Figure 57 illustrates the potential complementary functions of those languages.

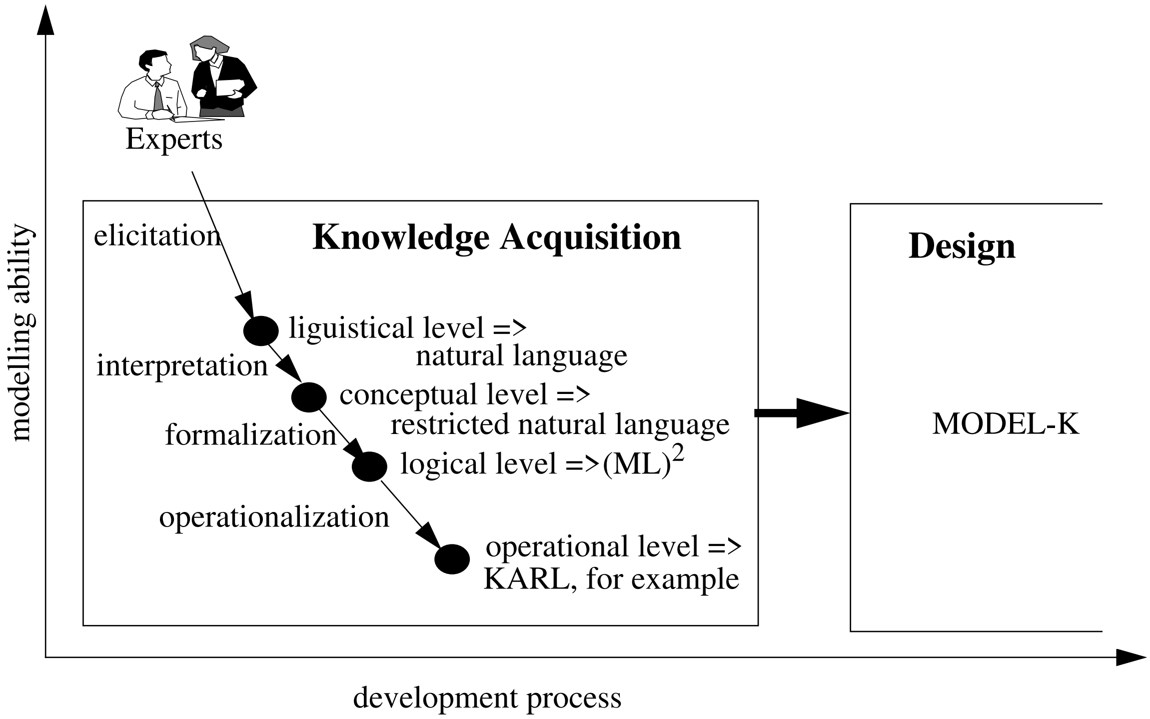


Figure 57. The happy family of the different languages.

Overview on MODEL-K, FORKADS, $(ML)^2$, and KARL

	MODEL-K	FORKADS	$(ML)^2$	KARL
Key Idea	***How to customize an existing AI shell?***	***How to combine full formal detail with executability?***	***How to use different types of logic to formalize models?***	***What can we be concluded from software engineering and information system development?***
(1). Domain layer	- Concept hierarchies; - relationship types and relationships; - concept instances; - extensional descriptions of relationships; and intensional descriptions of relationships.	- First-order logic; order-sorted.	- First-order logic; - order-sorted; - modularization.	- Classes, predicates, is-a and is-element relationship, functional and set-valued attributes, and multiple-attribute inheritance for modelling concepts and relationships; - extensional and intensional descriptions of concepts and relationships by rules; - constraints.

Overview on MODEL-K, FORKADS, $(ML)^2$, and KARL

	MODEL-K	FORKADS	$(ML)^2$	KARL
(2). Inference layer	- Inference actions; - roles; and - inference action bodies using Lisp, PROLOG, etc.	- Primitive inferences: terminological (sort-related) and referential (truth-conditions); - composed inferences by axioms or by procedural attachment in C.	- Primitive inference actions; - first-order logic for describing them; - roles.	- Elementary inference actions; - composed inference actions (levelled inference layer); - object-oriented Horn logic with stratified negation for describing inference actions; - roles (i.e., stores, views, and terminators).
(3). Task layer	- Sequence, branch and loop; and - subtasks.	- Procedural language with iteration over sorts, sequence, alternative, and subtasks; - inferences are seen as primitive tasks; - non-logical constructs (*use_axioms* and *assert*).	- *has-solution-pia*$_i$*()*; - *old-solution-pia*$_i$*()*; - *more-solution-pia*$_i$*()*; - *give-solution-pia*$_i$*()*; - sequence, loop, and branch of *give-solution-pia*$_i$*()*. - Variables storing all results of inferences.	- Sequence, branch, loop; and - hierarchical refinement (subtasks).
(4). Strategic layer	Implementation of reflective problem solving by modules with a whole model of expertise of an object system as domain layer.	No language primitives.	No language primitives.	No language primitives.

Overview on MODEL-K, FORKADS, $(ML)^2$, and KARL

	MODEL-K	FORKADS	$(ML)^2$	KARL
mapping (1)-(2)	- A role can be connected with one domain concept. - An inference action can be connected with one domain relation.	Homomorphism between sort structures.	-A lift definition maps domain terms on expressions at the inference layer. - Reflection rules connect derivations at the meta-layer with axioms of the domain layer.	Arbitrary view definitions by object-oriented Horn logic extended by stratified negation.
mapping (2)-(3)	It is possible to ask for the content of roles	Tasks return truth value and variable bindings.	Primitive inference actions have a truth value and return variable bindings.	The domain layer and inference layer language define a semantical structure for the task layer language
speciality	Allows to model reflective problem-solving processes and incremental development of an efficient expert system.	Procedural attachment as part of a formal language.	- Realizing inference layer and task layer as meta-layers, which allows the reasoning about arbitrary domain terms. - Provides logic as a representation formalism for control knowledge at the task layer.	- Applies object-orientation; - allows hierarchical structuring of complex models of expertise; - has a formal semantics; - is an operationalization and formalization language.

6.2 A Comparison with Software Engineering and Information System Development[1]

One reaction to the so-called "software crisis" in the late sixties was the development of informal, formal, and operational specification languages, first in the domain of software engineering, and later in information systems development. As there exists a large number of these languages, only an incomplete list is given here:[2]

- ATOL (cf. [BRA82]) is a BASIC-like operational specification language for text processing applications.
- EREA (cf. [DHL86a], [DHL86b], [DHR91]), the short-cut for Entity, Relation, Attribute, and Event, is a formal specification language for information systems based on extended temporal logic.
- FRORL (cf. [TWJ92]) is a formal and operational specification language which integrates frames and production rules by a nonmonotonic variant of Horn logic.
- Gist (cf. [BGW82], [Fea87]) is an operational specification language specifying a system by a set of states, stimulus-response rules, and constraints. In [CSB82] the idea of symbolic execution is presented as a technique to evaluate such specifications. [Dub89] presents a Gist-dialect which is based on nonstandard modal logic.
- INCOME (cf. [LNO89], [NOS92]) is a method for the development of information systems. It provides an executable specification language based on semantical data models similar to SHM (cf. [BrR84]) and high-level Petri nets, i.e. predicate-transition nets (cf. [GeM86]). This proposal is congenial to [Stu87] and [Kun89]. For a comparison of INCOME and KARL see [AFL92a].
- LOTOS (cf. [LMW90]) is a formal specification language for data communication protocols and services. Its static semantics are defined by an attributed grammar, and its dynamic semantics are based on algebraic concepts.
- The Modal Action Logic (MAL) (cf. [FiP87]) is a formal specification language which consists of four layers: (i) typed first-order logic, (ii) modal logic with actions and agents, (iii) a deontic logic, and (iv) an interval logic with temporal operators.
- The MODEL language describes the data and their interrelationship by equations without referencing any computer operations, i.e. it does not include any program control and program flow concepts (cf. [PrL89]).[3]
- OBJ (cf. [Shu89]) is an algebraic specification language. It consists of a hierarchy of objects in which each object declares an abstract data type, which corresponds to sorts, of operators, and of a set of equations, which define the meaning of an operator.

---

1. This section is an elaboration of the paper [FAL93]. Therefore, I would like to thank Jürgen Angele, Dieter Landes, and Rudi Studer for their helpful cooperation.
2. An overview and bibliography on algebraic specification can be found in [BKL91].
3. This language should not be confounded with MODEL/KADS (cf. [Bar93]).

- PAISLey (cf. [Zav82], [Zav91]) is an operation specification language which specifies a system by a set of function definitions.
- Often, PROLOG [ClM81] is proposed as an operational and formal specification language (cf. [LeS84], [Ven84], [Lee85], [Tav85], [Bra89]).
- PSA/PSL (cf. [TeH77], [Say90]) is one of the oldest software-specification language. It is based on the Entity-Relationship-Attribute (ERA) model and is used for the specification of information systems.
- The Prototyping System Description Language PSDL (cf. [KrR89]) is used together with modified dataflow diagrams as a graphical tool for rapid prototyping of system specifications.
- The Requirements Modelling Language (RML) (cf. [GBM86]) describes a software system in terms of entities, activities, and assertions and is based on temporal logic.
- RSL (cf. [BBD77], [Alf90]), also one of the oldest specification languages, is developed in the Software Requirements Engineering Program (SREM). It is a program flow-oriented language for specifying control flow.
- Structures Analysis (SA) (cf. [Ros77], [You89]) is a method which contains several informal instruments for software specifications like data dictionary, dataflow diagrams, and state-transition diagrams. There exist several approaches to formalizing or operationalizing these informal description. Teamwork/ES (cf. [BlH88], [LFF91]) is a prototype system that executes a real-time Structured-Analysis specification in an interactive and graphical manner. Other approaches recommend a combination with Petri nets (cf. [FrS91]), with algebraic specification techniques (cf. [FrD89]), or with functional programming languages (cf. [EMS89]).
- SEGRAS (cf. [Kra87]) is a formal and semigraphical language combining Petri nets and abstract data types for the specification of distributed systems.
- SETL (cf. [SDD86], [DFS89]) is an executable, procedural specification language. Besides the elementary data types, SETL provides sets, tuples, and maps. For example, a loop statement in SETL can be represented like:

  "loop forall x in S | P(x) do *statements* end forall;" [DFS89], p. 278.

  This program iterates over all elements x of the set S which satisfy P(x), and executes the *statements* for each iteration.
- SPECS (cf. [DGG87]) uses high-level Petri nets combined with object-oriented extensions for specifying complex distributed systems.
- SXL (cf. [Wan88]) is a high-level state-transition language which allows system specification by means of a model that is based on the finite-state machine.
- TROLL (cf. [JSH91], [HJS93]) is an object-oriented specification language for information systems based on first-order logic and temporal logic.
- The Vienna Development Method VDM (cf. [BjJ78], [BjJ82], [BJM87], [BMJ88]) is a method of developing software systems using formal specification languages like Meta-IV or VDL. A system is modelled by a collection of operations defined in terms of pre-conditions and post-conditions on an underlying state. The method has been used for a large variety of different problems over more than a decade.

- XS-2 (cf. [StS88]) is a language for specification and rapid prototyping of user interfaces.
- Z (cf. [Spi88], [Spi89], [BHL90]) is a formal specification language which describes the data, their interrelationship, and the functional behavior of a system by sets, relations, and functions.

Although this is an incomplete list (in fact it is more like the top of an iceberg) it is evident that a detailed and meaningful comparison of KARL with all these specification languages is beyond the scope of this thesis. Apart from obvious space problems, this comparison would have an essential and serious problem. A language like KARL, stemming from the domain of expert systems, uses quite a different conceptual model of the system, compared to the languages originating in the software-engineering domain. Therefore, a detailed comparison with these languages at the so-called "epistemological level", like the contrasting of KARL with MODEL-K, FORKADS, and $(ML)^2$, is not possible here.

As a consequence, a different strategy has been chosen. The conceptual model of KARL (i.e., the model of expertise), is compared here with the conceptual model of a standard technique in software engineering. To make the comparison technically well-founded, it is first shown how the conceptual model of a standard technique of software engineering based on informal specification instruments can be formalized by KARL. Structured Analysis (cf. [You89]) has been chosen here, and an operational and formal semantics has been defined for it by means of KARL (cf. [FAL93]), with the specification of a lending-library system by KARL serving as an example. Then, the different layers of a model of expertise of KARL will be compared with the various instruments of Structured Analysis.

6.2.1 How to Model Structured-Analysis Specifications with KARL

According to [You89] a system specification using Structured Analysis consists of the following components:

- A ***data dictionary*** and ***entity-relationship diagrams*** are used to describe the static system aspects, i.e. the structure and domains of the data used.
- ***Dataflow diagrams*** describe the functional behavior of a system. The whole system is decomposed into single processes, and their interaction is represented by sharing data. Dataflow diagrams mainly consist of processes, stores, and the dataflows between them. They can be hierarchically refined to allow the representation with different grain size.
- ***Process-specification*** techniques allow the description of the behavior of elementary processes which are not further refined by other dataflow diagrams.
- As dataflow diagrams should not be used to define control flow between processes, a separate instrument for representing the ***control flow*** between the processes is added.

6.2.1.1 The Static System Aspects: The Data Model

A ***data model*** describes the structure of the data which are used by a system. In Structured Analysis this aspect is covered by a data dictionary and entity-relationship diagrams. KARL provides L-KARL to describe these issues (cf. section 2.1). The expressive power of L-KARL highly exceeds the scope of simple entity-relationship diagrams. L-KARL can model single-valued (i.e., functional) and set-valued attributes, is-a relationships with attribute inheritance, domain restrictions and range restrictions for attributes, and arbitrary relationships. In addition, knowledge can be further described by logical clauses.

In other aspects, L-KARL does not cover all aspects of a data dictionary. In figure 58 an example for a data dictionary is given. An extension of L-KARL which is to meet the requirements of a data dictionary mainly consists of two components:

- The built-in type "string" must be refined by further sub-types.
- The modelling-primitive ***list*** must be introduced into L-KARL.

```
name = courtesy-title + first-name + (middle-name) + last name
courtesy-title = [Mr. | Miss | Mrs. | Ms. | Dr. | Professor]
first-name = {legal-character}
middle-name = {legal-character}
last-name = {legal-character}
legal-character = [A-Z | a-z | 0-9 | ´ | - | ]
```

Figure 58. An example for a data dictionary (cf. [You89], p. 191)

6.2.1.2 The Functional Behavior of a System: Dataflow Diagrams and Process Specification

In the following it is shown how KARL can be used to define stores, terminators, processes, and dataflows. For more details compare the sections 2.1 and 3.2.

Stores serve as containers. Stores can serve as input stores or output stores of a process. Input stores provide input to a process whereas results produced by a process are collected in an output store. Stores serve as passive elements which are filled with data. Class definitions define the structure of the data which are stored in it.

KARL distinguishes two kinds of ***terminators***. ***Views*** supply input to the system, and ***terminators*** receive output from the system. External data can be supplied by a user or a database, for example. If the structure of these external data does not fit the data structure used by the system, a mapping can be defined. The mapping transforms the data structure from the external to the internal format and vice versa. This aspect is not discussed in Structured Analysis. Class definitions of views and terminators define the structure of the data which are received from a view or delivered to a terminator.

A ***process*** is a system activity which transforms given input to output. Its ***external behavior*** is described by its input stores, views, output stores, and terminators. The execution of a process rewrites the contents of its output stores and terminators. The external description of a process is given in figure 59.

The ***internal behavior*** of a process is described declaratively by using rules and constraints. Again, these rules and constraints can be formulated by applying the logical language Logical-KARL (L-KARL). When using L-KARL to specify a process it is possible to express declaratively ***what*** a process should do without referring to ***how*** it is done.

The ***dataflow*** between the processes, stores, and terminators is specified by defining the premises and conclusions of processes. For the solution to most problems, it is necessary to use several cooperating processes. This ***process cooperation*** is established via stores, e.g. the output store of one process is used as input store by another one. The combination of several processes and stores is called a dataflow diagram, it can be designed and represented graphically. The ***structure of the data involved in a dataflow*** between a process and a store, a terminator, or a view is specified by the class definitions of the store, terminator, or view. The definitions specify the structure of the data which can be kept in the store, delivered by a view, or delivered to a terminator.

```
ELEMENTARY INFERENCE ACTION process_j
    PREMISES
        list of input stores and views
    CONCLUSIONS
        list of output stores and terminators
    DEFINITIONS
        list of auxiliary class and predicate definitions
    RULES
        list of rules
    NECESSARY CONDITIONS
        list of constraints
END
```

Figure 59. A process description

For easier handling of the complexity of a dataflow diagram for complex problems KARL provides ***levelled dataflow diagrams*** as it is proposed in [You89].

6.2.1.3 The Dynamic Behavior of a System: The Control Flow

"Now we look at a third kind of modeling tool, the ***state-transition diagram*** (also known as STD), which highlights the time-dependent behavior of a system." [You89] State-transition diagrams model the state of a system and their possible changes, i.e. the transition from one state to another one. Conditions or actions can be used to specify these transitions. With these means the control flow of a system is specified.

By means of Procedural-KARL (P-KARL) the software engineer can specify the control flow of a system with notions which are well-known from procedural languages, such as Pascal or Modula-2. One goal of KARL is its declarative character. This is achieved by applying dynamic logic (cf. [Koz90]) and its semantics for P-KARL. For details compare sections 2.2 and 3.3.

The graphical representation by program-flow diagrams shows the similarity of a P-KARL specification and a specification by a state-transition diagram. One of the main restrictions of P-KARL are the facts that it does not allow concurrency and that P-KARL programs are deterministic.

6.2.1.4 An Example

A lending library is modelled in KARL. A client asks for some books. If a book is in the store, the user obtains information on the location. If a book is lent out to another person, the name of that person is given. If a book is unknown to the system the user does not receive anything. The dataflow diagram is pictured in figure 60.

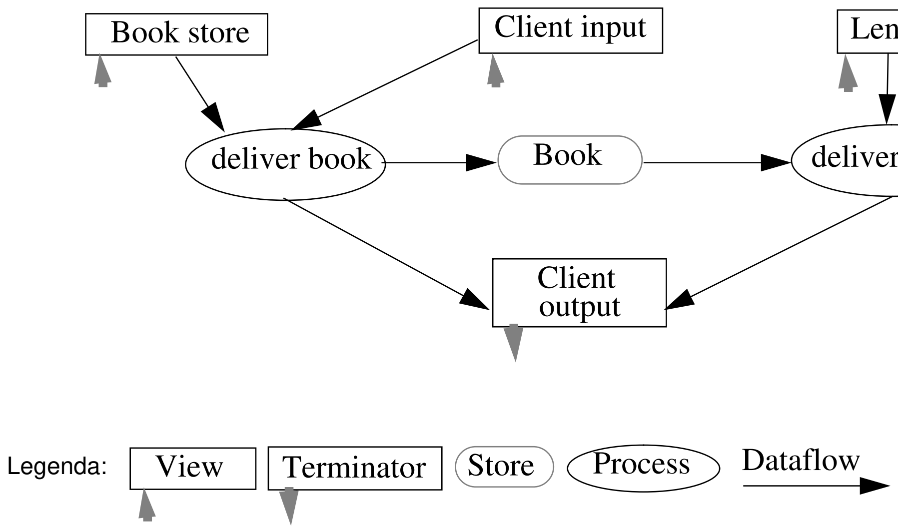


Figure 60. The dataflow diagram of the lending-library system

Alphabet of the inference layer:

$^{I}\Sigma$ = {available_book, bookstore, desired_book, lender, lent_book, not_available_book, not_found_book}

$^{I}A_{E\rightarrow}$ = {author, name, title}

$^{I}A_{E\rightarrow\rightarrow}$ = {lender, location}

Π = {found_book}

A_{Π} = {b}

X_E = {x_B, y_B}

X_V = {x_a, x_t}

^{I}V = {Book store, Client input, Lender file}

^{I}S = {Book}

^{I}T = {Client output}

EIA = {deliver book, deliver name}

```
STORE Book
    /* The store Book contains the desired books which cannot be found in the
    view Book store. */
    CLASS not_found_book
        ELEMENT_ATT
            title : {String};
            author : {String};
    END
END

TERMINATOR Client output
    /* The terminator Client output contains every book desired which can be
```

```
    found in the views Book store or Lender file. */
    CLASS available_book
        ELEMENT_ATT
            title : {String};
            author : {String};
            location :: {String};1
    END
    CLASS not_available_book
        ELEMENT_ATT
            title : {String};
            author : {String};
            lender :: {lender};2
    END
    CLASS lender
        ELEMENT_ATT
            name : {String};
    END
END

VIEW Book store
    /* The view Book store contains every book which is currently available in the lending
    library. */
    CLASS bookstore
        ELEMENT_ATT
            title : {String};
            author : {String};
            location :: {String};
    END
END

VIEW Client input
    /* The view Client input delivers the desired books. */
    CLASS desired_book
        ELEMENT_ATT
            title : {String};
            author : {String};
    END
END

VIEW Lender file
    /* The view Lender file contains all books which are lent out together with
    information about the lender. */
    CLASS lent_book
        ELEMENT_ATT
            title : {String};
            author : {String};
            lender :: {lender};
    END
    CLASS lender
        ELEMENT_ATT
            name : {String};
    END
END
```

1. A book can have several places, i.e. there exist several copies of the same book.
2. A book can have several lenders, i.e. there exist several copies of the same book.

ELEMENTARY INFERENCE ACTION deliver book
/* *deliver book* matches the input of the client with the content of the book store. The location of the books which can be found are written into the terminator *Client output* (together with the title and author of the book). The desired books which cannot be found are put in the store *Book*. */
PREMISES
Book store, Client input;
CONCLUSIONS
Book, Client output;
DEFINITIONS
PREDICATE found_book
b : {desired_book};
END
RULES
/* Books which can be found are placed in the terminator *Client output* and their location is indicated. */
$\forall x_a, \forall x_B, \forall x_t, \forall y_B$ (x_B ε available_book ←
x_B[title:x_t,author:x_a] ε bookstore ∧
y_B[title:x_t,author:x_a] ε desired_book).
/* Books which cannot be found are placed in the store *Book*. */
$\forall x_a, \forall x_B, \forall x_t, \forall y_B$ (found_book(b : y_B) ←
x_B[title:x_t,author:x_a] ε bookstore ∧
y_B[title:x_t,author:x_a] ε desired_book).
$\forall y_B$ (y_B ε not_found_book ←
y_B ε desired_book ∧
¬found_book(b : y_B)).
END

ELEMENTARY INFERENCE ACTION deliver name
/* *deliver name* matches the books of the store *Book* with the content of the view *lender file*. Books which are lent out are written to the terminator *Client output* (together with information about the lender). */
PREMISES
Book, Lender file;
CONCLUSIONS
Client output;
RULES
/* Books which are lent out are placed in *Client output* together with the names of the persons who have borrowed a copy of them.*/
$\forall x_a, \forall x_B, \forall x_t, \forall y_B$ (x_B ε not_available_book ←
x_B[title:x_t,author:x_a] ε lent_book ∧
y_B[title:x_t,author:x_a] ε not_found_book).
END

After defining the data structure, the dataflow, and the processes, the ***control flow*** between the processes must be defined:

Alphabet of the task layer:[1]
F = {deliver book, deliver name}
X = {Book, o}
PS = {∅(Book, not_found_book)}

1. Views and terminators do not have a semantics at the task layer, they are just syntactical sugar. Therefore, they are not a part of the alphabet.

```
(STORES: Book; TERMINATORS: Client output) :=
    deliver book(VIEWS: Book store, Client input);
IF ¬∅(Book,not_found_book)
THEN
    (TERMINATORS: Client output) :=
        deliver name(STORES: Book; VIEWS: Lender file)
ENDIF
```

6.2.2 The Relation Between the Different Layers of KARL and the Different Techniques of Structured Analysis

A KARL specification consists of three layers. The domain layer specifies the domain-specific knowledge which is used by the problem-solving method. The dataflows, the elementary steps of a problem-solving method, and the use of the domain knowledge for problem-solving are specified at the inference layer. The task layer is used to define the sequence of the steps of a problem-solving method. Before modelling the example presented in the last subsection, we had the following conceptual view on the relationships of these layers and the different instruments of Structured Analysis: We thought that the domain layer corresponds to the data model (i.e., data dictionary and entity-relationship diagrams), the inference layer corresponds to the dataflow diagrams and process specifications, and the task layer corresponds to the state-transition diagrams. But then we realized that the specification of the lending-library system ended up with an empty domain layer, which motivated us to take a closer view at the relationship of these layers of KARL and instruments of Structured Analysis.[1]

6.2.2.1 Task Layer = State-Transition Diagrams?

The task layer of KARL and the state-transition diagrams of Structured Analysis are roughly correspondent because both are means of specifying the control flow of a system.

6.2.2.2 Inference Layer = Functional Specification by Dataflow Diagrams?

[Sch92a] compares Structured Analysis and KADS and ends up with three distinctions between dataflow diagrams with process specifications on the one side and an inference layer on the other side:[2]

1. This is again a significant example how a formal knowledge level language supports an appropriate conceptualization of a phenomenon (cf. section 5.3.3).

2. A fourth distinction made by [Sch92a] is the statement that inference structures in KADS cannot be refined whereas dataflow diagrams can be hierarchically refined. “An inference structure can be viewed as a data-dependency diagram at the lowest level of functional decomposition.” [Sch92a], p. 165. In our opinion, it is necessary to introduce hierarchical refinement of inference structures if KADS is used to specify complex systems, as it was the case in KARL inference structures.

- "A first difference between the functional description in KADS and in conventional software engineering is that the data elements in inference structures do not refer directly to elements of the data model ..." [Sch92a], p. 165
- "A second difference is that the KADS knowledge sources are not described via detailed process specifications." [Sch92a], p. 165
- "A third distinction is that KADS provides a ***typology*** of basic functions." [Sch92a], p. 165

The second and the third difference do not seem to be of an essential quality, especially if one considers formal descriptions of knowledge sources (i.e., inference actions) in $(ML)^2$ (cf. [BaA92b]) and considers that an appropriate typology of inference actions is rather an continuing research effort than a given result (cf. [Abe92a], [Abe92b]). The first distinction, however, points out a difference of inference structures and dataflow specifications. The input data, intermediate data, and output data of a problem-solving process in KADS are handled by so-called roles or meta-classes.

> "A ***meta-class*** description serves a dual purpose, (i) it acts as a ***placeholder*** for domain objects, describing the role that these objects play in the problem solving process, and (ii) it points to the ***type(s)*** of the domain objects that can play this role." [WSB92], p. 19

Roles connect an inference layer with a domain layer. Inputs and outputs of inference actions do not directly refer to domain expressions. In addition, in KADS a domain view defines another connection between inference layer and domain layer.

> "The ***domain view*** specifies how particular parts of the domain theory can be used as a >>body of knowledge<< by the knowledge source." [WSB92], p. 20

As already discussed in section 3.2.2 of the thesis, KARL contains three different types of roles modelling the different tasks of roles and domain views.

- ***Views***: A view reads knowledge and data from the domain layer which serve as input to an inference action.[1]
- ***Terminators***: A terminator writes results of the problem-solving process back to the output-data part of the domain layer.
- ***Stores:*** An intermediate role of an inference layer collects output from inference actions and serves as input to other inference actions. Such a role has no connection to the domain layer. They are instruments of modelling the dataflow dependencies between inference actions.

When comparing these three types of roles with the means of dataflow diagrams, the difference becomes obvious. Stores in KARL correspond to stores in Structured Analysis. Views used to read input data from the domain layer and terminators roughly correspond to what is called terminators in Structured Analysis. But in Structured Analysis there is ***no*** instrument which corresponds to ***views*** which are applied to read knowledge from a domain layer. Thus, KARL inference structures correspond to dataflow diagrams but extend them by mapping rules in view bodies which are used to read knowledge for the inferencing process. This distinction of dataflow diagrams and inference structures necessarily leads to the third equation. Is the

1. This role type corresponds to what was called meta-relation in an older version of KARL (cf. [FAL91]).

domain layer identical to the data model in Structured Analysis?

6.2.2.3 Domain Layer = Data Model?

Before modelling the lending-library system in KARL, we had a clear conceptual idea how the different layers of KARL are related to the different means of Structured Analysis. But then it turned out that the KARL-specification ended up with an empty domain layer! The terminology, i.e. the syntactical structure of the dataflows, are defined by the class definitions of the roles at the inference layer.

This motivated us to take a closer view at the purpose of a domain layer in KADS, and it became obvious that ***the domain layer does not have a direct counterpart in Structured Analysis***. The domain layer of a knowledge-based system is used to model the domain-specific knowledge which is necessary to solve the task:

> "Domain knowledge can be viewed as a declarative theory of the domain. In fact, adding a simple deductive capability would enable a system in theory (but, given the limitations of theorem-proving techniques, not in practice) to solve all problems solvable by the theory." [WSB92], p. 19.

The inference layer and task layer contain additional knowledge which allows an efficient solution of a specific task. But already the domain layer has to contain enough ***knowledge*** to solve the task in principle, i.e. to produce the desired effect. As Structured Analysis is not designed to specify ***knowledge-based*** systems there is no direct correspondence to the domain layer. Therefore, the domain layer in the example of section 6.2.1.4 is empty. The data model of Structured Analysis specifies the structure of the data which are used and produced by the processes. The "problem-solving behavior" of a system is specified by the dataflow diagrams, the process specifications, and the state-transition diagrams. These instruments do not just specify an effective solution of the problem but they are used to ***model*** the effective and efficient solution of the problem.

The inference layer of KARL models the main parts of Structured Analysis. The stores, terminators (i.e., terminators and views), processes, dataflows, and structure of the data, i.e. the data model, are defined at the inference layer. The data model is defined by an alphabet and by the class definitions and predicate definitions of the stores, views, terminators, and processes. It is not defined by a separate instrument as in Structured Analysis. The task layer specifies the sequence of these processes and corresponds to state-transition diagrams in Structured Analysis.

6.2.3 Conclusion: On the Difference Between Conventional Software and Knowledge-Based Systems

The comparison of KARL and Structured Analysis has ended up in the surprising result that the most significant distinction between the two approaches is contained in the domain layer. It is to contain all domain-specific knowledge which, in principle, enables the knowledge-based system to solve the given tasks. The inference layer and task layer just add additional knowledge to improve the efficiency of the problem-solving process. The extension of dataflow diagrams by views in KARL inference layers is just a consequence of this difference as views are instruments for supplying the inference layer with this domain knowledge.

Thus the difference of a conceptual model of a conventional software system oriented on Structured Analysis on the one hand and a conceptual model of a knowledge-based systems oriented on the model of expertise of KADS on the other hand boils down in the difference in the domain layer, which does not have a counterpart in Structured Analysis.

KARL and Structured Analysis

Model of Expertise	KARL	Structured Analysis
Domain Layer:	Specification of domain knowledge and domain terminology.	No correspondence in Structured Analysis.
Inference Layer:	• Terminology used by the problem-solving method; • dataflow between inferences; • hierarchically levelled refinement of composed inference actions; • specification of elementary inferences; • definitions of domain views.	• Domain terminology; • dataflow between processes; • hierarchically levelled refinement of composed processes; • specification of elementary processes; • no correspondence in Structured Analysis.
Task Layer:	Specification of control flow.	Specification of control flow.

Conclusion

Part one of this conclusion will give a summary of the highlights of KARL. As the main feature of KARL is the integration of formalization and operationalization of models of expertise into one language, a few problems connected with this integration will be discussed in part two, which is concerned with limitations of KARL. Then, there will be a short sketch of the relationship between KARL and machine learning, and of the relationship between KARL and software reuse.

C.1 The Highlights of KARL

[AFS90] dealt with the question of what the knowledge engineer could learn from the software engineer. As an answer to this question, the knowledge specification language KARL was developed as a means for the building process of knowledge-based systems. Experiences in software engineering showed the usefulness of formal and executable specification languages which are a means of overcoming shortcomings of life cycle-oriented process models. The logical language used at the domain layer and inference layer is strongly influenced by semantical data models, object-oriented data models, and deductive data bases. The inference layer and the task layer are strongly influenced by the dataflow diagram, i.e. the structured analysis, and procedural specification languages from the domain of software engineering.

By introducing these ideas into knowledge engineering it was possible to resolve an “old” debate in knowledge acquisition, i.e. how to reconcile the symbol level with the knowledge level. KARL is a language which fills the gap between these levels.

- KARL is related to the symbol level because it is a language with a formal and operational semantics.

- KARL is related to the knowledge level because it abstracts from all implementational details which are concerned with the realization by a specific computer program, and because it provides the epistemological primitives which have been identified to be necessary for the description of knowledge at the knowledge level.

The main features of KARL are:

- KARL provides ***epistemologically*** adequate modelling primitives allowing knowledge specifications at the “knowledge level”. The model of expertise as defined in the KADS-I project was used as guideline for these modelling primitives.
- KARL is a ***formal*** knowledge specification language, i.e. it has a declarative semantics.
- KARL is an ***operational*** knowledge specification language which allows prototyping.

Additional features are:

- Complex specifications, required for real-life systems, quickly become difficult to oversee. Therefore, ***hierarchical refinement*** are provided.
- Most of KARL´s modelling primitives have a ***graphic*** representation to allow KARL specifications to be used as a medium of communication.

Besides KARL, some other languages have been developed during the last few years with the objective of formalizing or operationalizing KADS models of expertise. Apart from all technical details, the main difference between KARL and the other languages is that KARL aims to be a ***formal and operational*** knowledge specification language. The objective here is not only operationalization in principle, i.e. the possibility to use a kind of theorem-proving, but also effective and efficient evaluation to allow quick prototyping. This causes some problems which will be discussed next.

C.2 Limitations of KARL

First, two limitations of the current version of KARL are given. Then, a general limitation of KARL concerning its operationalization is discussed.

C.2.1 Current Limitations of KARL

KARL is not able to deal with real-time applications, i.e. it cannot be applied to ***monitoring tasks***, and it cannot be used to specify ***interactive support systems***. The control flow at the task layer is fixed, and it is not possible to interrupt the problem-solving process. KARL does not allow to specify any kind of external interruption of the reasoning process of a system. In section 3.6 of this thesis a restricted version of the model of cooperation is introduced into KARL. The user can provide input to the system, he can receive output from the system, and he can provide control-flow information for it. But the system is always the active agent. It asks for information or delivers information. It is not possible to model the user as an active agent who determines the moment of cooperation by interrupting the reasoning process of the system. This is a significant drawback for the possibility of modelling highly interactive systems which supports a human decision process. KARL always assumes the system to be the agent and the user to be a subordinate entity.

In addition, even the term of ***model of cooperation*** as introduced by [GrB92] is misleading because it assumes that two agents of the same status cooperate to solve a given task. But a more appropriate view would present human agents using a ***tool*** (i.e., the system) which supports him by solving the task ([Flo87], [FRS89]). Humans do not cooperate with their tools, but they make use of them to reach their goals (cf. [DaF93]).

These restrictions of KARL reflect a common practice in knowledge engineering but should be overcome by future versions of KARL.

C.2.2 Actual Limitation of KARL

In this subsection a general limitation of an operationalization of KARL is discussed, using the specification of the elementary inference action *Check* of chronological backtracking (cf. section 5.1.2.2) as an illustration:

/* *Check* checks whether some states are a possible solution. */
ELEMENTARY INFERENCE ACTION ***Check***
    PREMISES
        Goal_States, Selected_States;
    CONCLUSIONS
        Possible_Solutions;
    RULES
        /* Every state which fits the goal predicate is a possible solution. */
        $\forall x_{St}$
            $(x_{St}\ \varepsilon\ \textit{possible\_solutions} \leftarrow$
                $x_{St}\ \varepsilon\ \textit{selected\_states} \wedge$
                $\textit{goal}(g : x_{St})).$
END;

A domain view *Goal_States* would define the mapping of this predicate for a specific domain:

VIEW ***Goal_States***
    DEFINITIONS
        PREDICATE ***goal***
                g : {selected_states};
        END;
    RULES
    $\forall x_{St}\ (\textit{goal}(g : x_{St}) \leftarrow ...).$
END;

The above given clauses describe declaratively "***what***" the elementary inference action is doing. The first clause states that a selected state is checked whether it is a goal state, i.e. whether the goal predicate is true for it. If it is a goal state it is considered as a possible solution.

The clauses of the mapping would describe the goal predicate declaratively.

A naive operationalization of this inference action is the computation of its perfect Herbrand model. It contains all goal states, i.e. all states for which the goal predicate holds. Therefore, one single evaluation of this one elementary inference action has the computational complexity of computing all possible solutions for the given problem, because the extension of the goal predicate is exactly the set of all states for which the problem is solved. The runtime of the entire problem-solving method is therefore much longer than the runtime of a problem-solving method which computes all or only one solutions in one step. More elaborated evaluation techniques (cf. [Ull88], [Ull89], and [Ang93]) can counteract this effect and allow prototyping with reasonable runtimes. This kind of prototyping can be used to check whether the modelled solution is able to solve the problem. However, it can never be used to compare the computational complexity of several specified problem-solving methods. It is always possible, that the most primitive problem-solving method which computes a solution brute-forth has a less computational complexity than a more sophisticated problem-solving method. Computational complexity must be considered in the design phase of knowledge engineering. Only an efficient implementation of the elementary inference actions allows proper estimations of the runtime. A declarative description of what a single step does can be used to simulate the behaviour of this step by computing its minimal model, but it ***should never be mixed*** with an efficient description of how this single step is done.

Besides this general remark, two main distinctions exist between formal and operational KARL:

- Operational-KARL restricts the expressive power of L-KARL because it requires finite models for sets of clauses and definitions. This does not change the semantics but the expressiveness of L-KARL.
- Due to optimization of the evaluation of an elementary inference action not the entire well-typed perfect Herbrand model is computed but only the subset used to fill its output stores. Therefore, not the perfect Herbrand model of the elementary inference action is checked for well-typing but only the subset which defines the new content of the output stores. This slightly changes the semantics of a call of an elementary inference action but improves its computation.

For more details see [Ang93].

C.3 KARL and Machine Learning

The formal specification of knowledge-based systems requires a high amount of effort and skill from the knowledge engineer. An alternative way is specifying the behaviour of the desired program by examples and automatically generating a formal specification (cf. [Jia92], who discusses a method for generating software specifications from examples). This seems to be a very useful approach, if the knowledge consist of large data bases describing problems and their solutions. The generalization of formal descriptions from examples has been discussed under several headings:[1]

- Generating software specifications from example (cf. [BMW84], [Jia92]).
- Automatic programming (cf. [Bal85], [Sum77]).
- Inductive logic programming (cf. [Mug92]).
- Knowledge discovery in data bases (cf. [PSF91], [Yag92]).
- Machine learning (cf. [ShD90]).

There exists a lot of experience in applying machine learning procedures for knowledge acquisition. Procedures like ID3 (cf. [Qui84]) or AQ (cf. [MMH86]), for example, can be used to learn decision trees or production rules. Systems which can be used to learn first-order expressions are BLIP (cf. [Mor87b]) or DISCIPLE (cf. [KoT87]), for example. In KARL, such procedures can be applied for several purposes, e.g.:

- They can be used to construct a domain theory from examples, i.e. to build up a hierarchy of classes (is-a relationships) and class definitions (i.e. domain restrictions and range restrictions in class definitions) from a set of examples and rules describing these classes.
- The logical description of an inference action or its mapping can be learned using examples (i.e., input-output pairs) as input.

[FeK91], [FeK92], [FKN93], [Fen93] and [FeW93] describe the learning procedures RELAX and JoJo, which extract rules from examples. They have been applied to large data sets (i.e. sets with more than 10,000 examples) or data sets which have been constructed with the expert system shell D3 (cf. [PuG92]). They can work with nominal, ordinal, continuous, and multi-valued attributes and handle noise and unknown values in data sets. In [FGS93] a case study considering the application of these algorithms to a knowledge acquisition problem is reported. A set of case data is used by JoJo to learn a set of rules for a heuristic knowledge base.

Currently, these algorithms cannot be directly applied to KARL specifications because they use a different knowledge representation formalism (i.e. attribute-value pair vectors). In principle, two possibilities exist to fill this gap:

- In [LDG91] it is discussed how knowledge which is represented in a restricted first-order logic language can be transformed and represented such that it can be used by such algorithms.
- FOIL (cf. [Qui90]) is an algorithm using the main ideas of the above discussed algorithms but it applies them directly to a heuristic search process in a first-order logic representation formalism.

Machine-learning algorithms can be used to improve the modelling process in KARL, the epistemological distinctions of KARL can also improve the application of machine-learning algorithms for knowledge acquisition.

First, the distinction of different types and kinds of knowledge in KARL help to define ***different types*** of learning tasks and a ***smaller grain size*** for them then common in machine learning where a complete reasoning system is to be learned by examples.

The following ***different types*** of knowledge can basically be distinguished by the modelling primitives of KARL:

1. Also data compression can be viewed in this context.

- A machine learning algorithm can be used to learn or refine an entire domain theory.
- It can be used to construct a mapping from a given inference action onto a given domain layer.
- It can be used to learn an inference action for a given domain and mapping.
- It can be used to learn a procedural control flow at the task layer.

Also, generic problem-solving methods or knowledge-level models of a task can be applied to distinguish ***different kinds*** of knowledge, and therefore ***different kinds*** of learning tasks, and again a ***smaller grain size*** for them. [Cla85] introduced a model of heuristic classification as a general frame work for diagnostic processes. It mainly consists of three steps: a data abstraction, i.e., concrete values are mapped on intervals, a heuristic match from abstract problem descriptions to abstract solutions, and the hierarchical refinement of these solutions until a final solution has been found.

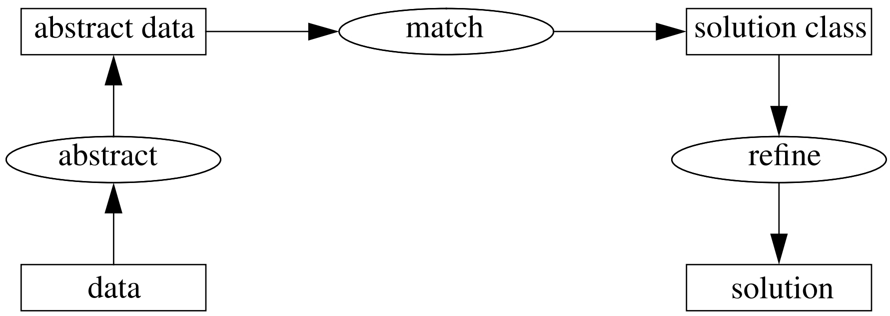


Figure 61. Inference structure of heuristic classification

Comparing this knowledge level model of heuristic classification with machine learning it becomes obvious that these three quite different steps collapse into a single one. Neither decision trees as produced by ID3 (cf. [Qui84]) nor production rules produced by an AQ algorithm (cf. [MMH86]) make these distinctions. They learn a classifier which immediately tries to classify given objects by their attribute values. The same problem appeared, when the algorithm JoJo was applied to learn a set of heuristics rules for a knowledge base (cf. [FGS93]), using cases as an input. Instead of learning a network of rules which abstract attribute values, match these abstract description with a diagnosis, and refine them, just a flat set of rules was produced which had attribute values as premises and final diagnoses as conclusions.

> "The simplest problem-solving methods for classification are decision tables and decision trees ..." ([PuG92], p. 553)

The application of a knowledge level model of a given task can improve the application of machine learning algorithms to knowledge acquisition problems in two directions:

- The computational complexity of a learning task is reduced when it is split into subtasks.
- Different learning tasks become visible as target for different steps. To give an example: learning rules which abstract attribute values from given ones is a type of learning task which tries to construct new attributes by given ones, whereas the matching of abstract attribute values to classes of diagnoses is in fact a classification task.

Thus, both the different epistemological primitives of KARL and the knowledge level descriptions which apply them can improve the application of machine learning to knowledge acquisition problems.

C.4 KARL and Software Reuse

“Ask not what you can do
for your software
but what your software can do
for you.“
[Tra88b], p. 30

“From the beginning, ***software reuse*** has been touted as a means for overcoming the software crises. ... In spite of its promise, software reuse has failed to become standard practice for software construction.” [Kru92], p. 132f. In [Kru92] different approaches to software reuse are analyzed and compared, with the following criteria emerging:

- ***Abstraction***. It is identified as a key issue of software reuse. If an implemented code is the only description of a component, reuse normally requires more effort than reimplementation does. The purpose of abstraction is to narrow the cognitive distance between the mental concept of a system and its implementation, i.e. its implemented semantics.
- ***Selection***. If a library of reusable components exists, the user needs support in selecting the appropriate component for solving his specific task.
- ***Specialization***. In general, generic components must be adapted to a specific domain and task to solve the given problem.
- ***Integration***. In general, only a combination of reusable components solves the entire problem of a user.

In the following section, the four categories are used to analyze and compare eight different categories of software reuse:[1]

High-level languages: C is an example of such high-level languages. The expressions and statements in a high-level language define an abstract machine by grouping and integrating a list of assembler statements into one expression. Programming languages are the most commonly used approach of software reuse. It is a level of abstraction on which software reuse is a common practice. Selection of reusable components is relatively simple because of the limited number of constructs available in the languages. The problem of high-level languages is the fact that there is still a significant cognitive distance between an informal mental model

1. Cf. [Bör89] for a bibliography on software reuse and [Tra88a] for a provocative discussion of some myths about software reuse.

of the system and its implemented semantics. High-level languages define a low level of abstraction only.

Ad-hoc reuse of existing code fragments: This is also a common practice of software reuse. An abstract description of the reused components often exists in the mental model of the programmer only. There is no systematic way and no tool support for this kind of software reuse. Selection, specialization, and integration depend on the intuition of the programmer and on the presence of already existing appropriate code fragments.[1]

Source code libraries: Examples of source-code libraries are trigonometrical and statistical libraries, such as SPSS and SAS. Functions like *sin(x)* have a common and mathematically well-defined meaning. The name *sin* is sufficient to understand the meaning of the implemented function. This kind of software reuse has been successful in application areas where such application-specific abstractions can be used to index software components. Other examples of this kind of reuse are object-oriented languages such as Smalltalk, with large libraries of reusable components. The greater part of the effort in working with these languages is not implementing new solutions but finding the already existing appropriate components. Experiences with these libraries show that they are helpful tools if they are restricted to specific components such as mathematical functions and statistical routines. General-purpose libraries would have to be very large, and they would be difficult to use.

Software schemes: This approach uses code components that are accompanied by a formal semantic description. Selection, instantiation (i.e., specialization), and integration of such components can be supported by automatic tools. The formal description of these components abstracts from implementational details and therefore has a less cognitive distance between the conceptual level and the implemented code. These formal descriptions of abstract algorithms are automatically instantiated to executable source code. However, formal "specifications for schema abstraction can be large and complex. Even with automated tools it can be difficult for software developers to locate, understand, and use schemes." [Kru92], p. 156

Application generators: Application generators produce executable code from specifications. These generators are specialized on a very narrow type of task. As a result of this limited range, specification languages can incorporate be very high-level constructs (e.g., templates and graphics) and task-specific which allow even non programmers to produce executable code.

Libraries of several application generators do not yet exist. In [Kru92], MOLE and SALT (cf. [Mar88]) are used as examples for application generators. MOLE and SALT are method-specific knowledge acquisition tools. Other examples of application generators are parser generators such as Lex and Yacc. Application generators reduce significantly the cognitive distance between the conceptual model of a system and its implementation, and automate the software-design process to a high degree. In application generators, a specification at the conceptual level is mapped on an executable one automatically. The main problem with application generators is that they have a narrow application area. It is difficult (i.e., it is a contradiction in adjection) to build application generators that can be applied to a broad range of tasks.

Very-high-level languages: Very high-level languages (VHLL) are executable like high-level

1. "In the worst cases, a software developer spends more time locating, understanding, modifying, and debugging a scavenged code fragment than the time required to develop the equivalent software from scratch." [Kru92], p. 141

languages but they aim to increase the level of abstraction. "The primary concern in VHLLs is not efficiency in program execution but rather efficiency in implementing and modifying programs." (cf. [Kru92], p. 165) In VHLLs, the automatic code-generation is similar to that of application generators. Otherwise, the VHLLs are application-independent general-purpose languages of higher complexity (e.g., they use first-order logic). KARL is an example for such a VHLL. The main advantages of VHLLs is that high-level abstractions with a mathematical semantics are automatically mapped on executable code.

- The abstractions of VHLLs have a less cognitive distance between the conceptual level and the language level than those of HLLs, but are suitability for general-purpose software development as HLLs. However, the run-time performance of VHLL specifications is often poor.
- The abstractions of VHLLs have a higher cognitive distance between the conceptual level and the language level than application generators. However, VHLLs are not bound to a narrow type of tasks.

Therefore, VHLLs define an intermediate level between HLLs and application generators.

Transformational systems: This approach applies a two-step procedure. First, a system specification is constructed using a specification language, which could even be at a higher level of abstraction than a VHHL. Second, this specification is transformed into an efficient executable system. The second step can be supported by tools, and might but require manual effort from the software developer (cf. [Smi90]). In [Lan93] it is discussed how a system specification in KARL can be used to create a design model, i.e. how such a KARL specification can be transformed into an efficiently executable system description.

Software architectures. "Reusable software architectures are large-grain software frameworks and subsystems that capture the global structure of a software system design." [Kru92], p. 173. Examples in the domain of expert systems are the rule-based and blackboard architectures, which are used to build up several expert systems. This approach generalizes the application generator approach.

Reuse is also an important topic in the current discussions in the knowledge engineering community.[1] Different approaches are discussed in the research literature:

The most popular approach with several applications is the ***role-limited method*** approach, i.e the ***application generator*** approach. For example, MOLE, SALT, SIZZLE (cf. [Mar88]), D3/ CLASSIKA (cf. [PuG92]), or PROTÉGÉ (cf. [Mus89b]) are knowledge acquisition tools and expert-system shells with a fixed problem-solving method. These expert-system shells can be applied to a fixed type of tasks. The problems of this approach are:

- A high effort is required to develop a knowledge acquisition tool for every problem-solving method, i.e. for every type of task.
- Tasks that require the combination of several problem-solving methods are not supported.

To overcome these shortcomings of application generators, [MDK92], [Mus92], [ChJ93], and [Ste92] propose libraries of reusable methods or generic tasks. These methods have a finer

1. Cf. the workshop on reusable problem-solving methods or mechanism at the 7th Banff Knowledge Acquisition for Knowledge-Based Systems Workshop, October, 1992.

grain size than conventional expert system shells. A complete problem-solving process must be modelled by several methods. These approaches are analogous to the ***source-code libraries*** idea in software engineering. The main characteristics of these approaches is that currently these methods are only described by code and by informal descriptions of the code. Therefore, there is little support for the selection, specialization, and integration of these methods. The lack of descriptions that abstract from implementational details, but do not lead to imprecise natural-language descriptions, makes it difficult to compare such methods, and to provide a precise description of their problem-solving ability. Mathematical and statistical libraries have been successful, because their components have a well-defined mathematical meaning that allows developers to address such components by their names. Currently, there exist neither exhaustive libraries of reusable problem-solving methods nor experience in their reuse for knowledge engineering.

The KADS group proposes a library of ***interpretation models***. [BWS87] define a set of such models. In addition, a finer grain size of reuse in the set of ***standard primitive inference actions*** of which a knowledge engineer can define new problem-solving methods. As these models or inference actions did have neither an executable nor a formal semantics they represent a kind of reuse at the conceptual level without direct support by implemented components. In order to improve this approach, the formal knowledge-specification language $(ML)^2$ (cf. [HaB92]) has been developed and applied to the specification of several problem-solving methods and of primitive inference actions ([Abe92a], [Abe92b], [BaA92b]). But even these formalizations do not lead to executable models.

According to the different levels of reuse, KARL comes into this discussion at four of them. First, KARL is a ***very-high level language***. Second, in [Lan93] it is discusses how KARL specifications can be transformed into design models and efficient implementations. This activity, which is subject of current research, is an example of the ***transformational-system*** approach.

Third, in chapter five of the thesis the description of reusable problem-solving methods/ mechanisms in KARL is given. These descriptions have two main advantages:

- The source code is described at the conceptual level in terms of domain knowledge, inferences, control flow, etc.
- The source code is described in a formal manner by a language that has a formal semantics, but abstracts from all details which are concerned with a specific implementation.

That way, KARL improves the ***source code library*** approach. The code components are described at a higher level of abstraction. The implemented code blocks can be used as black boxes, because KARL provides an informal (i.e., graphical) and a formal description of it. Note that, the KARL specification does not replace the implemented component. The specification defines the meaning of the components at a conceptual and formal level. For the implementation of the code blocks, a programming language such as C can be used.

Finally, KARL can improve reusability in the ***software schema*** approach. A library of interpretation models formalized in KARL corresponds to the schema approach. Two main task must be done to fulfil this purpose:

- The appropriate interpretation models must be identified and formalized in KARL.

- The KARL specifications must be examined systematically for preconditions, postconditions, invariants, and temporal relationships that hold for the interpretation models.[1]

This properties can be used to support the selection, specification, and integration of such specifications. A formal description of a given task can be used to automatically derive the appropriate interpretation models from the library as suggested by [KRT89] in the PARIS approach.[2]

The different kinds of reuse which are enabled with formal and operational knowledge specification languages like KARL is an interesting research topic for future efforts.

> “What are the open areas of research in software reuse? There is clearly much work that remains to be done. In general, the search for high-level abstractions for software artifacts is probably the most crucial, but this type of research is not new or unique to software reuse.“
>
> [Kru92], p. 179

1. The substantial effort that is required for formal proofs is justified if the software will be reused several times.
2. See also [YKS85].

Bibliography

[Abe92a] M. Aben: On the Specification of Knowledge Model Components. In *Proceedings of the 7th Knowledge Acquisition for Knowledge-Based Systems Workshop KAW´92*, Banff, Canada, October 11-16, 1992.

[Abe92b] M. Aben: A Formalisation of KADS Inferences, research report, ESPRIT Project P5248 KADS-II, KADS-II/T1.2/TR/UvA/40/1.0, University of Amsterdam, November, 1992.

[Abe92c] M. Aben: Guidelines for the Formal Specification of KADS Models of Expertise, research report, ESPRIT Project P5248 KADS-II, KADS-II/T1.2/TR/UvA/37/1.0, University of Amsterdam, November, 1992.

[ABW88] K. R. Apt, H. A. Blair, and A. Walker: Towards a Theory of Declarative Knowledge. In Minker (ed.), *Foundations of Deductive Databases and Logic Programming*, Morgan Kaufman Publ., Los Altos, 1988, pp. 89-148.

[AFL91a] J. Angele, D. Fensel, D. Landes, and R. Studer: An Assignment Problem in Sisyphus - No Problem with KARL. In [Lin92c].

[AFL91b] J. Angele, D. Fensel, D. Landes, and R. Studer: Explorative Prototyping with KADS. In *Proceedings of the Workshop Knowledge Acquisition: From Science to Technology to Tools, 9th National Conference on AI (AAAI-91)*, Anaheim, July 15th, California, 1991.

[AFL91c] J. Angele, D. Fensel, D. Landes, and R. Studer: KARL: An Executable Language for the Conceptual Model. In *Proceedings of the 6th Knowledge Acquisition for Knowledge-Based Systems Workshop KAW´91*, Banff, Canada, October 6-11, 1991.

[AFL91d] J. Angele, D. Fensel, D. Landes, S. Neubert, and R. Studer: Knowledge Engineering in the Context of Related Fields of Research. In O. Herzog et al. (eds.), *Text Understanding in LILOG: Integrating Computational Linguistics and Artificial Intelligence,* Springer-Verlag, Lecture Notes in Artificial Intelligence, no 546, Berlin, 1991.

[AFL92a] 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, pp. 23-39.

[AFL92b] J. Angele, D. Fensel und D. Landes: An executable model at the knowledge level for the office-assignment task. In [Lin92d].

[AFL93] J. Angele, D. Fensel, D. Landes, S. Neubert, and R. Studer: Model-based and Incremental Knowledge Engineering: The MIKE Approach. In J. Cuena (ed.), *Proceedings of the IFIP TC12 Workshop on Artificial Intelligence from the Information Processing Perspective - AIFIPP´92*, Madrid, Spain, 14-15 September, 1992, Elsevier Science Publisher B.V., Amsterdam, to appear 1993, pp. 139-168.

[AFS90] J. Angele, D. Fensel, and R. Studer: Applying Software Engineering Methods and Techniques to Knowledge Engineering. In D. Ehrenberg et al. (eds.), *Wissensbasierte Systeme in der Betriebswirtschaft, Reihe betriebliche Informations- und Kommunikationssysteme, no 15*, Erich Schmidt Verlag, Berlin, 1990, pp. 285-304.

[AFS92] J. Angele, D. Fensel, and R. Studer: Formalizing and Operationalizing Models of Expertise: A Basis for Validation. In *Pre-Proceedings of the IFIP TC12 Workshop on Artificial Intelligence from the Information Processing Perspective - AIFIPP´92*, Madrid, Spain, 14-15 September, 1992.

[AFS93] J. Angele, D. Fensel, and R. Studer: A Models of Expertise in KARL. Institut für Angewandte Informatik und Formale Beschreibungsverfahren, University of Karlsruhe, reserach report, no 272, March 1993.

[AHS93] H. Akkermans, F. van Harmelen, G. Schreiber, and B. Wielinga: A Formalisation of Knowledge-Level Models for Knowledge Acquisition, *International Journal of Intelligent Systems, Special Issue Knowledge Acqisition as Modeling,* part II, no 2, vol 8, 1993, pp. 169-208.

[AKS93] S. Aitken, O. Kühn, N. Shadbolt, and F. Schmalhofer: A Conceptual Model of Hierarchical Skeletal Planning and its Formalization. In *Proceedings of the 3rd KADS Meeting*, Munich, March 8-9, 1993.

[Alf90] M. Alford: SREM at the Age of Eight; the Distributed Computing Design System. In R. H. Thayer et al. (eds.), *System and Software Requirements Engineering*, IEEE Computer Society Press, Washington, 1990, pp. 392-402.

[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: Prototyping mit KARL (Prototyping with KARL), in preparation, 1993 (in German).

[AnK92] J. Angele and D. Küpper: Modula-2 an Alternative to C? In *ACM SIGPLAN Notices*, vol 27, no 4, April 1992, pp. 17-26.

[Apt90] K. R. Apt: Logic Programming. In J.v. Leeuwen (ed.), *Handbook of Theoretical Computer Science*, Elsevier Science Publisher B.V., Amsterdam, 1990, pp. 493- 574.

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

[BaA92a] J. R. Balder and H. Akkermans: TheMe: an Environment for Building Formal KADS-II Models of Expertise, *AI Communications*, vol 5, no 3, September 1992, pp. 136-147.

[BaA92b] J. R. Balder and J. M. Akkermans (eds.): Formal Methods For Knowledge Modelling in the COMMONKADS Methodology, A Compilation, Netherlands Energy Research Foundation ECN, report no ECN-C--92-080, 1992.

[BaG79] R. Balzer and N. Goldman: Principles of Good Software Specification and their Implications for Specification Language. In *Proceedings of Reliable Software (SRS)*, Boston, Massachusetts, April 1979, pp. 58-67.

[BaH91] F. Baader and P. Hanscke: A Schema for Integration Concrete Domains into Concept Languages, In *Proceedings of the 12th International Joint Conference on AI (IJCAI´91)*, Sydney, Australia, August 24-30, 1991, pp. 452-457.

[Bal85] R. Balzer: A 15 Year Perspective on Automatic Programming, *IEEE Transaction on Software Engineering*, vol 11, no 11, 1985, pp. 1257-1267.

[Bar93] M. Barbuceanu: MODEL/KADS: Building KADS Models of Expertise in a Term Classification Language, to appear in *Knowledge Acquisition*, 1993.

[BaS89] J. Bachant and F. Soloway: The Engineering of XCON, *Communications of the ACM*, vol 32, no 3, March 1989, pp. 311-318.

[BBC91] B. Bartsch-Spörl, B. Bredeweg, C. Coulon, U. Drouven, F. v. Harmelen, W. Karbach, M. Reinders, E. Vinkhuyzen, and A. Voß: Studies and Experiments with Reflective Problem Solvers. In research report, ESPRIT-Basic Research Project P3178 REFLECT, IR.3.1,2, RFL/BSR-UvA/II.2/1, August 1991.

[BBD77] T. E. Bell, D. C. Bixler, and M. E. Dyer: An Extendable Approach to Computer-Aided Software Requirements Engineering, *IEEE Transactions on Software Engineering*, vol 3, no 1, January 1977, pp. 49-60.

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

[BeH89] J. Bell and R. J. Hardiman: The Third Role - the Naturalistic Knowledge Engineer. In D. Diapper (ed.), *Knowledge Elicitation - Principles, Techniques and Applications*, John Wiley & Sons, New York, 1989, pp. 49-85.

[Ben56] H.D. Benington: Production of Large Computer Programs. In *Proceedings of the ONR Symposium on Advanced Programming Methods for Digital Computers*, June 1956, pp. 15-27.

[Ber87] D. C. Berry: The Problem of Implicit Knowledge, *Expert Systems*, vol 4, no 3, August 1987, pp. 144-151.

[BGW82] R. M. Balzer, N. M. Goldman, and D. S. Wile: Operational Specifications as the Basis for Rapid Prototyping, *ACM SIGSOFT Software Engineering Notes*, vol 7, no 5, December 1982, pp. 2-16.

[BHA93] J. Balder, F.v. Harmelen, and M. Aben: A KADS/(ML)^2 Model of a Scheduling Task. In J. Treur et al. (eds.), *Formal Specification of Complex Reasoning Systems*, Proceedings of the ECAI-92 Workshop on Formal Specification Methods for Complex Reasoning Systems, European Conference on Artificial Intelligence (ECAI-92), Wien, Austria, August 3-7, 1992, Ellis Horwood, Chichester, to appear 1993.

[BHL90] D. Bjorner, C. A. R. Hoare, and H. Langmaack (eds.): *VDM´90. VDM and Z - Formal Methods in Software Development*, Lecture Notes in Computer Science, no 428, Springer-Verlag, Berlin, 1990.

[Bic92] P. Bickel: Untersuchung zweier Ansätze der Wissensakquisition am Beispiel der umwelttechnischen Beratung von Betrieben (A Consultation System for Companies Concerning Ecological Requirements. A Comparison of Two Knowledge Acquisition Approaches), master thesis, Institut für Angewandte Informatik und Formale Beschreibungsverfahren, University of Karlsruhe, 1992 (in German).

[BjJ78] D. Bjorner and C. B. Jones: *The Vienna Development Method: The Meta-Language*, Lecture Notes in Computer Science, Springer-Verlag, no 61, Berlin, 1978.

[BjJ82] D. Bjorner and C. B. Jones: *Formal Specifications & Software Development*, Prentice-Hall Int., Englewood Cliffs, 1982.

[BJM87] D. Bjorner, C. B. Jones, M. Mac Airchinnigh, and E. J. Neuhold: *VDM´87. VDM - A Formal Method at Work*, Lecture Notes in Computer Science, no 252, Springer-Verlag, Berlin, 1987.

[BKL91] M. Bidoit, H.-J. Kreowski, P. Lescane, F. Orejas, and D. Sannella (eds.): *Algebraic System Specification and Development*, Lecture Notes in Computer Science, no 501, Springer-Verlag, Berlin, 1991.

[BlH88] R. Blumofe and A. Hecht: Executing Real-Time Structured Analysis Specifications, *ACM SIGSOFT Software Engineering Notes*, vol 13, no 3, July 1988, pp. 32-39.

[BLN86] C. Batini, M. Lenzerini, and S. B. Navathe: A Comparative Analysis of Methodologies for Database Schema Integration, *ACM Computing Surveys,* vol 18, no 4, December 1986, pp. 323-364.

[BMJ88] R. Bloomfield, L. Marshall, and R. Jones (eds.): *VDM´88. VDM - The Way Ahead*, Lecture Notes in Computer Science, no 328, Springer-Verlag, Berlin, 1988.

[BMW84] A. Borgida, J. Mylopoulos, and H. K. T. Wong: Generalization/Specialization as a Basis for Software Specification. In Brodie et al. (eds.), *On Conceptual Modelling*, Springer-Verlag, Berlin, 1984, pp. 87-114.

[Boe86] B.W. Boehm: A Spiral Model of Software Development and Enhancement, *ACM SIGSOFT Software Engineering Notes*, vol 11, no 4, August 1986, pp. 14-24.

[Boe88] B.W. Boehm: A Spiral Model of Software Development and Enhancement, *IEEE Computer,* May 1988, pp. 61-72.

[Böh92] I. Böhme: Operationalisierung von KARL (Operationalization of KARL), master thesis, Institut für Angewandte Informatik und Formale Beschreibungsverfahren, University of Karlsruhe, 1992 (in German).

[Bör89] J. Börstler: Wiederverwendbarkeit und Softwareentwicklung - Probleme, Lösungsansätze und Bibliographie (Reuse and Software Development - Problems, Solutions, and Bibliography), *SE Softwaretechnik Trends*, vol 9, no 2, September 1989, pp. 62-76 (in German).

[BRA82] W. P. Bodd, P. Ramsay, T. H. Axford, and D. G. Parkyn: A Prototyping Language for Text-Processing Applications, *ACM SIGSOFT Software Engineering Notes*, no 5, vol 7, December 1982, pp. 50-52.

[Bra83] R. J. Brachman: What IS-A Is and Isn´t: An Analysis of Taxonomic Links in Semantic Networks, *IEEE Computer*, vol 16, no 10, October 1983, pp. 30-36.

[Bra85] R. J. Brachman: On the Epistemological Status of Semantic Networks. In R. Man et al. (eds.), *Readings in Knowledge Representation*, Morgan Kaufman Publ., Los Altos, California, 1985, pp. 191-215. Originally appeared in: N. V. Findler (ed.), *Associative Networks: Representation and Use of Knowledge by Computers*, Academic Press, New York, 1979.

[Bra87] R. J. Brachman: The Myth of the One True Logic, *Computer Intelligence*, vol 3, no 3, August 1987, pp. 168-172.

[Bra89] I. Bratko: Fast Prototyping of Expert Systems Using Prolog. In G. Guida et al. (eds.), *Topics in Expert Systems Design*, Elsevier Science Publisher B. V., North-Holland, 1989, pp. 69-86.

[Bro86] A. G. Brooking: The Analysis Phase in Development of Knowledge-Based Systems. In W. A. Gale (ed.), *AI and Statistic*, Addison-Wesley Publishing Company, Reading, Massachusetts1986, pp. 321-334.

[BrR84] M. L. Brodie and D. Ridjanovic: On the Design and Specification of Database Transactions. In M. L. Brodie et al. (eds.), *On Conceptual Modelling. Perspectives from Artificial Intelligence, Databases, and Programming Languages*, Springer-Verlag, Berlin, 1984, pp. 277-312.

[BrS85] R. J. Brachman, and J. G. Schmolze: An Overview of the KL-ONE Knowledge Representation System, *Cognitive Science*, vol 9, no 2, April 1985, pp. 171-216.

[BrW84] J. A. Breuker and B. J. Wielanga: Techniques for Knowledge Elicitation and Analysis, report no 1.5 ESPRIT project 12, University of Amsterdam, Department of Social Science Informatics and Laboratory for Experimental Psychology, July 1984.

[BrW89] J. Breuker and B. Wielinga: Models of Expertise in Knowledge Acquisition. In G. Guida et al. (eds.), *Topics in Expert Systems Design*, Elsevier Science Publisher B. V., North-Holland, 1989, pp. 265-295.

[BRW90] B. Bredeweg, M. Reinders, and B. Wielinga: GARP: A Unified Approach to Qualitative Reasoning, report VF-memo 117, University of Amsterdam, 1990.

[Bun90] A. Bundy (ed.): *Catalogue of Artificial Intelligence Techniques*, 3rd ed., Springer-Verlag, Berlin, 1990.

[BüW92] S. Bürsner and Th. Wetter: An Operational KADS Modelling Language and Tool Support for its Application. In *Proceedings of the 7th Banff Knowledge Acquisition for Knowledge-Based System Workshop (KAW´92)*, Banff, Canada, October 11-16, 1992.

[BWS87] J. Breuker, B. Wielinga, M.v. Someren, R. de Hoog, G. Schreiber, P. de Greef, B. Bredeweg, J. Wielemaker, and J.-P. Billault: Model-Driven Knowledge Acquisition: Interpretation Models. In ESPRIT project P1098, report, University of Amsterdam, 1987.

[ByC88] T. Bylander and B. Chandrasekaran: Generic Tasks for Knowledge-Based Reasoning: the “Right” Level of Abstraction for Knowledge Acquisition. In B. Gaines et al. (eds.), *Knowledge Acquisition for Knowledge Based Systems*, Knowledge-Based Systems, vol 1, 1988, Academic Press, London, pp. 65-77.

[CGT89] S. Ceri, G. Gottlob, and L. Tanca: What You Always Wanted to Know About Datalog (And Never Dared to Ask), *IEEE Transactions on Knowledge and Data Engineering*, vol 1, no 1, March 1989, pp. 146-166.

[Cha87] B. Chandrasekaran: Towards a Functional Architecture for Intelligence Based on Generic Information Processing Tasks. In *Proceedings of the 10th International Joint Conference on AI (IJCAI´87)*, Milan, Italy, August 23-28, 1987, pp. 1183-1192.

[Che76] P.P. Chen: The Entity Relationship Model. Toward a Unified View of Data, *ACM Transactions on Database Systems*, vol 1, 1976, pp. 9-36.

[ChJ93] B. Chandrasekaran and T.R. Johnson: Generic Tasks and Task Structures: History, Critique and New Directions. In J.-M. Daveid et al. (eds.), *Second Generation Expert Systems*, Springer-Verlag, Berlin, to appear 1993.

[CJS92] B. Chandrasekaran, T.R. Johnson, and J.W. Smith: Task-Structure Analysis for Knowledge Modeling, *Communications of the ACM*, vol 35, no 9, September 1992, pp. 124-137.

[Cla78] K. L. Clark: Negation as Failure. In H. Gallaire et al. (eds.), *Logic and Data Bases*, Plenum Press, New York, 1978, pp. 293-322.

[Cla83] W. J. Clancey: The Epistemology of a Rule-Based Expert System - a Framework for Explanation, *Artificial Intelligence*, vol 20, 1983, pp. 215-251.

[Cla85] W.J. Clancey: Heuristic Classification, *Artificial Intelligence*, vol 27, 1985, pp. 289-350.

[Cla86] W. J. Clancey: From GUIDON to NEOMYCIN and HERACLES in Twenty Short Lessons: ORN Final Report 1979-1985, *The AI Magazine*, August 1986, pp. 40-60.

[Cla89] W.J. Clancey: The Knowledge Level Reinterpreted: Modeling How Systems Interact, *Machine Learning*, vol 4, 1989, pp. 285-291.

[Cla92a] W. J. Clancey: Model Construction Operators, *Artificial Intelligence*, vol 53, no 1, 1992, pp. 1-115.

[Cla92b] W.J. Clancey: The Knowledge Level Reinterpreted: Modeling in Socio-Technical Systems, *International Journal of Intelligent Systems, Special Issue Knowledge Acquisition as Modeling,* part I, no 1, vol 8, 1993, pp. 33-50.

[ClM81] W. Clocksin and C. Mellish: *Programming in Prolog*, Springer-Verlag, Berlin, 1981.

[Cor89] E. S. Cordingley: Knowledge Elicitation Techniques for Knowledge-Based Systems. In D. Diapper (ed.), *Knowledge Elicitation - Principles, Techniques and Applications*, John Wiley & Sons, New York, 1989, pp. 89-176.

[Cru92] I. F. Cruz: DOODLE: A Visual Language for Object-Oriented Databases. In *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data*, San Diego, California, June 2-5, 1992, pp. 71-80.

[CSB82] D. Cohen, W. Swartout, and R. Balzer: Using Symbolic Execution to Characterize Behavior, *ACM Sigsoft Software Engineering Notes*, vol 7, no 5, 1982, pp. 25-32.

[CPV89] T. Christaller, F. di Primo, and A. Voß: *Die KI-Werkbank BABYLON* (The KI-Tool BABYLON), Assison Wesley, Bonn, 1989 (in German).

[DaF93] M. Daniel and G. Fieguth: Computergestütztes Problemlösen und Verbesserung der Wissenskommunikation als humanorientierte Gestaltungsaufgabe (Computer-supported Problem Solving and Improvment of Knowledge Communication Developed by a Human-centred Method). In *Proceedings of Arbeit und Technik*, January 27-28, Bonn, 1993, Teubner Verlag, Stuttgart, to appear 1993 (in German).

[DaH88] M. Davis and S. Hakiel: Knowledge Harvesting: A Practical Guide to Interviewing, *Expert Systems*, vol 5, no 1, February 1988, pp. 42-50.

[DFS89] E. Dubinsky, S. Freudenberger, E. Schonberg, and J. T. Schwartz: Reusability of Design for Large Software Systems: An Experiment with the SETL Optimizer. In T. J. Biggerstaff et al. (eds.), *Frontier Series: Software Reusability - vol I - Concepts and Models*, ACM Press, New York, 1989, pp. 275-293.

[DGG87] J. Dähler, P. Gerber, H.-P. Gisiger, and A. Kündig: A Graphical Tool for Design and Prototyping of Distributed Systems, *ACM SIGSOFT Software Engineering Notes*, vol 12, no 3, 1987, pp. 25-35.

[DHL86a] E. Dubois, J. Hagelstein, E. Lahou, F. Ponsaert, A. Rifaut, and F. Williams: The ERAE Model: A Case Study. In T. W. Olle et al (eds.), *Information Systems Design Methodologies: Improving the Practice*, Elsevier Science Publishers B.V., North-Holland, 1986.

[DHL86b] E. Dubois, J. Hagelstein, E. Lahou, F. Ponsaert, A. Rifaut, and A. Rifaut: A Knowledge Representation Language for Requirements Engineering, *Proceedings of the IEEE*, vol 74, no 10, October 1986, pp. 1431-1444.

[DHR91] E. Dubois, J. Hagelstein, and A. Rifaut: A Formal Language for the Requirements Engineering of Computer Systems. In A. Thayse (ed.), *From Natural Language Processing to Logic for Expert Systems*, John Wiley & Sons, Chichester, 1991.

[Dij76] E.W. Dijkstra: *A Discipline of Programming*, Engelwood Cliffs, N.J., Prentice-Hall, 1976.

[Din66001] German Industry Norm (DIN), no 66001.

[DKV92] U. Drouven, W. Karbach, and Angi Voß: Solving the Office-Allocation Task in Reflective MODEL-K. In [Lin92d].

[Dor90] M. Dorfman: System and Software Requierements Engineering. In R. H. Thayer and M. Dorfman (eds.): *System and Software Requierements Engineering*, IEEE Computer Society Press, Los Alamitos, California, 1990, pp. 4-16.

[DrD86] H. Dreyfus and S. Dreyfus: *Mind over Machine*, The Free Press, New York, 1986.

[Dub89] E. Dubois: A Logic of Action for Supporting Goal-Oriented Elaborations of Requirements. In *Proceedings of the 5th International Workshop on Software Specification & Design*, Pittsburgh, USA, May, 19-20, 1989, pp. 160-168.

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

[Emd91] W. Emde: *Modellbildung, Wissensrevision und Wissensrepräsentation im Maschinellen Lernen* (Model Construction, Theory Revision, and Knowledge Representation in Machine Learning), Informatik-Fachbericht, no 281, Springer-Verlag, Berlin, 1991.

[EmK76] M. H. v. Emden and R. A. Kowalski: The Semantics of Predicate Logic as a Programming Language, *Journal of the Association for Computing Machinery*, vol 23, no 4, October 1976, pp. 733-742.

[EMS89] S. Eisenbach, L. McLoughlin, and C. Sadler: Data-Flow Design as a Visual Programming Language. In *Proceedings of the 5th International Workshop on Software Specification & Design*, Pittsburgh, USA, May, 19-20, 1989, pp. 281-283.

[End72] H. B. Enderton: *A Mathematical Introduction to Logic*, Academic Press, Boston, 1972.

[EST92] H. Erikson, Y. Shahar, S. W. Tu, A. R. Puerta, and M. A. Musen: Task Modeling with Reusable Problem-Solving Methods. In *Proceedings of the Seventh Banff Knowledge Acquisition for Knowledge-Based Systems Workshop*, Banff, Alberta, October 11-16, 1992.

[FAL91] D. Fensel, J. Angele, and D. Landes: KARL: A Knowledge Acquisition and Representation Language. In *Proceedings of the 11th International Conference on Expert Systems and their Applications, vol.1, General Conference Tools, Techniques and Methods*, 27-31 May, Avignon, 1991, pp. 513-528.

[FAL93] D. Fensel, J. Angele, D. Landes, and R. Studer: Giving Structured Analysis Techniques a Formal and Operational Semantics with KARL, *Proceedings of Requirements Engineering ´93 - Prototyping -*, Bonn, April 25 - 27, 1993, Teubner Verlag, Stuttgart, to appear 1993.

[FBA93] K. M. Ford, J. M. Bradshaw, J. R. Adams-Webber, and N. M. Agnew: Knowledge Acquisition as a Constructive Modeling Activity, *International Journal of Intelligent Systems*, *Special Issue Knowledge Acqisition as Modeling,* part I, no 1, vol 8, 1993, pp. 9-32.

[Fea87] M. S. Feather: Language Support for the Specification and Development of Composite Systems, *ACM Transactions on Programming Languages and Systems*, no 2, vol 9, April 1987, pp. 198-234.

[FeH93] D. Fensel and F.v. Harmelen: A Comparison of Languages which Operationalize and Formalize KADS Models of Expertise, in preparation.

[Fei80] E. Feigenbaum: Knowledge Engineering - The Applied Side of Artificial Intelligence, Stanford Heuristic Programming Project Report, Stanford University, 1980.

[FeK91] D. Fensel and J.Klein: A New Approach to Rule Induction and Pruning. In *Proceedings of the International Conference on Tools for Artificial Intelligence ICTAI´91*, San Jose, CA, November 10-13, 1991.

[FeK92] D. Fensel and J. Klein: RELAX, H-RELAX, I-RELAX: Three Algorithms fur Rule Induction and Pruning, research report, Instituts für Angewandte Informatik und Formale Beschreibungsverfahren, University of Karlsruhe (TH), no 235, Januar 1992.

[Fen92] D. Fensel: Knowledge Acquisition and the Interpretative Paradigm. In F. Schmalhofer et al. (eds.), *Contemporary Knowledge Egineering and Cognition*, *First Joint Workshop*, Kaiserslautern, Germany, February 21-22, 1991, Lecture Notes in Artificial Intelligence, no 622, Springer-Verlag, Berlin, Juli 1992, pp. 78-95.

[Fen93] D. Fensel: JoJo: Integration of Generalization and Specialization. In *Proceedings of the Workshop Knowledge and Data Engineering*, Atelier d´Ingenierie des Connaissances et des Donees, A.I.C.D., Strasbourg, France, January 25-27, 1993.

[FeS92] D. Fensel and R. Studer: An Analysis of Languages which Operationalize and Formalize KADS Models of Expertise. In *Proceedings of the 7th Knowledge Acquisition for Knowledge-Based Systems Workshop KAW´92*, Banff, Canada, October 11-16, 1992.

[Fet88] J. H. Fetzer: Program Verification: The Very Idea, *Communications of the ACM*, vol 31, no 9, September 1988, pp. 1048-1063.

[FeW93] D. Fensel und M. Wiese: Incremental Refinement of Rule Sets with JoJo, *Proceedings of the European Conference on Machine Learing ECML-93*, Vienna, Austria, April 5-8, 1993, Lecture Notes in Artificial Intelligence, no 667, Springer-Verlag, Berlin, 1993.

[FGS93] D. Fensel, U. Gappa, Schefe: A Machine Learning Procedure Applied to a Knowledge Acquisition Problem, submitted 1993.

[Fic87] S. Fickas: Automating the Analysis Process: An Example. In *Proceedings of the International Workshop on Software Specification and Design*, Monterey, CA, April 3-4, 1987, pp. 58-67.

[FiK85] R. Fikes and T. Kehler: The Role of Frame-Based Representation in Reasoning, *Communication of the ACM* , vol 28, no 9, September 1985, pp. 904-920.

[FiP87] A. Finkelstein and C. Potts: Building Formal Specifications Using “Structured Common Sense”. In *Proceedings of th 4th International Workshop on Software Specification and Design*, Monterey, California, April 3 - 4, 1987, pp. 108-113.

[FKN93] D. Fensel, J Klein, and U. Neubronner: RJ: An Environment for Learning from Example, *Proceedings of the 13th International Conference “Artificial Intelligence, Expert Systems, Natural Language”*, Avignon´93, May 24-28, 1993.

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

[Flo87] C. Floyd: Outline of a Paradigm Change in Software Engineering. In G. Bjerknes et al. (eds.), *Computers and Democracy - a Skandinavian Challenge*, Dower Publisher Comp., Aldershot, Hampshire, 1987, pp. 192-210.

[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, pp. 293-310.

[FRS89] C. Floyd, F.-M. Reisin, and G. Schmidt: STEPS to Software Development With User. In *Proceedings of the 2nd European Software Engineering Conference ESEC´89*, September 11-15, Warwick, Lecture Notes in Computer Science, no 387, Springer-Verlag, Berlin, 1989, pp. 48-64.

[FrS91] H. Franzen and G. Siegel: Die Methode der Strukturierten Analyse mit Petri-Netzen (SA/PN) als Echtzeiterweiterung (Extending Structured Analysis by Petri Nets for the Specifications of Real-Time Aspects). In *Proceeedings Requirements Engineering´91*, Marburg, *Informatik-Fachberichte*, no. 273, 1991, pp. 178-190.

[GBM86] S. J. Greenspan, A. Borgida, and J. Mylopoulos: A Requirements Modeling Language and its Logic, *Information Systems*, no 1, vol 11, 1986, pp. 9-23.

[Gei92] J. Geidel: An Environment for Modelling and Solving Optimisation Problems, In *Proceddings of the 2nd IFIP WG 7.6. - Conference on Optimization-Based Computer Aided Modelling and Design,* Schloß Dagstuhl, Germany, 28th September - 1st October, 1992.

[GeM86] N. Gehani and A. D. Mc Gettrick (eds.): *Software Specification Techniques*, Addison-Wesley, Wokingham, 1986.

[Gen86] H. J. Genrich: Predicate/Transition Nets. In W. Brauer et al. (eds.), *Petri Nets: Central Models and*

Their Properties, Advances in Petri Nets, 1986, Part I, Proceedings of an Advanced Course, Bad Honnef, September 8-19th, 1986, Lecture Notes in Computer Science, no 254, Springer-Verlag, Berlin, 1986, pp. 207-247.

[GrB92] P. de Greef and J. A. Breuker: Analysing System-User Cooperation in KADS, *Knowledge Acquisition*, vol 4, no 1, March 1992, pp. 89-108.

[Güh92] A. Gühnemann: Spezifikation eines wissensbasierten Systems zur Problemdefinition bei Umweltverträglichkeitsprüfungen für die Verkehrwegeplanung (Spezification of a Knowledge-Based System for Umweltverträglichkeitprüfung of Traffic Lines), master thesis, Institut für Angewandte Informatik und Formale Beschreibungsverfahren, University of Karlsruhe, 1992 (in German).

[HaB92] F. v. Harmelen and J. Balder: $(ML)^2$: A Formal Language for KADS Conceptual Models, *Knowledge Acquisition*, vol 4, no 1, March 1992, pp. 127-161.

[Har79] D. Harel: *First-Order Dynamic Logic*, Lecture Notes in Computer Science, vol 68, Springer-Verlag, Berlin, 1979.

[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.

[Heg1812] G. W. F. Hegel: *Wissenschaft der Logik*. In *Hegel´s Science of Logic*, translated by A. V. Miller, H. D. Lewis (ed.), Humanities Press International, Atlantic Highlands, NJ, 1993. Originally appeared 1812.

[HeR91] O. Herzog and C.-R. Rollinger (eds.): *Text Understanding in LILOG*, Lecture Notes in Artificial Intelligence, no 546, Springer-Verlag, Berlin, 1991.

[HJK89] F. R. Hickman, J. L. Killin, L. Land, T. Mulhall, D. Porter, and R. M. Taylor: *Analysis for Knowledge-Based Systems: a Practical Guide to the KADS Methodology*, Ellis Horwood Limited, Chichester, GB, 1989.

[HJS93] T. Hartmann, R. Jungclaus, and G. Saake: Spezifikation von Informationssystemen als Objektsysteme: Das TROLL-Projekt (Spezification of Information Systems as Object Systems: The TROLL-Project), *Emisa Forum*, no 1, 1993, pp. 2-18 (in German).

[HoK91] D. Hofbauer and R.-D. Kutsche: *Grundlagen des maschinellen Beweisens* (Introduction in Automatic Theorem-Prooving), 2nd edition, Friedrich Vieweg Verlagsgesellschaft, Braunschweig, 1991 (in German).

[HoN92] U. Hoppe and S. Neubert: Using Hypermedia for Integrating Mediating Representations in the Model-based Knowledge Engineering. In *Proceedings of the Workshop Knowledge Acquisition at the 9th National Conference on Artificial Intelligence AAAI'92*, San Jose, California, July 12-17, 1992.

[HoU79] J. E. Hopcroft and J. D. Ullmann: *Introduction to Automata Theory, Languages and Computation*, Addison-Wesley, 1979.

[HuK87] R. Hull and R. King: Semantic Database Modeling: Survey, Applications, and Research Issues, *ACM Computing Surveys*, vol 19, no 3, September 1987, pp. 201- 260.

[HWB92] F. van Harmelen, B. Wielinga, B. Bredeweg, G. Schreiber, W. Karbach, M. Reinders, A. Voß, J. M. Akkermans, and B. Bartsch-Spörl, and E. Vinkhuyzen: Knowledge-Level Reflection. In B. Le Pape et al. (eds.), *Enhancing the Knowledge-Engineering Process - Contributions from ESPRIT*, Elsevier Science Publ., B. V., Amsterdam, 1992.

[HWL83] F. Hayes-Roth, D. Waterman, and D. Lenat (eds.): *Building Expert Systems*, Addison-Wesley, Reading, London, 1983.

[Jia92] L. Jian: A Method of Aquiring Formal Specifications From Examples, *ACM SIGSOFT Software Engineering Notes*, vol 17, no 1, Januar 1992, pp. 52-56.

[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, pp. 211-229.

[JSH91] R. Jungclaus, G. Saake, T. Hartmann, C. Sernadeas: Object-Oriented Specifications of Information Systems: The TROLL Language. In Informatik-Bericht 91-04, Technical University Braunschweig (TU), 1991.

[JSV91] W. Jonker, J.W. Spee, L. in ´t Veld, and M. Koopman: Formal Approaches Towards Design in SE and Their Role in KBS Design. In *Proceedings of the IJCAI´91 Workshop on Software Engineering for Knowledge-Based Systems*, Sydney, Australia, August 24th, 1991.

[KaL90] W. Karbach and M. Linster: *Wissensakquisition für Expertensysteme. Techniken, Modelle und Softwarewerkzeuge* (Knowledge Acquisition for Expert Systems: Techniques, Models, and Tools), Carl Hanser Verlag, München, 1990 (in German).

[KaV92] W. Karbach and A. Voß: Reflecting About Expert Systems in MODEL-K. In *Proceedings of Expert Systems and their Applications, 12th International Workshop, vol 1 (Scientific Conference)*, June 1-6, Avignon, 1992, pp. 141-152.

[KaV93] W. Karbach and A. Voß: MODEL-K for Prototyping and Strategic Reasoning at the Knowledge Level. In J.M. David et al. (eds.), *Second Generation Expert Systems*, Springer-Verlag, Berlin, to appear 1993.

[KBD90] G. Klinker, C. Bhola, and G. Dallemagne, D. Marques, and J. McDermott: Usable and Reusable Programming Constructs. In *Proceedings of the 5th Knowledge Acquisition for Knowledge-Based Systems Workshop*, Banff, Canada, November 4-9, 1990. Also appeared in *Knowledge Acquisition*, vol. 3, 1991, pp. 117-135.

[Kee89] S. E. Keene: *Object-Oriented Programming in Common Lisp*, Addison-Wesley, Reading, 1989.

[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.

[Kid87] A. L. Kidd (ed.): *Knowledge Acquisition for Expert Systems. A Pratical Handbook*, Plenum Press, New York, 1987.

[KiL89] M. Kifer and G. Lausen: F-Logic: A Higher-Order Language for Reasoning about Objects, Inheritance, and Scheme. In *ACM SIGMOD Proceedings of the 18th International Conference on Management of Data*, Portland, Oregon, 31th May - 2th June 1989, pp. 134-146.

[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.

[KiW93] M. Kifer and J. Wu: A Logic for Programming with Complex Objects. To appear in *Journal of Computer and Systems Science*, 1993.

[KLS91] O. Kühn, M. Linster, and G. Schmidt: Clamping, COKAM, KADS, and OMOS: The Construction and Operationalization of a KADS Conceptual Model. In *Proceedings of the 5th European Knowledge Acquisition Workshop EKAW'91*, Crieff, Scotland, May 20-24, 1991, M. Linster et al. (eds.), GMD-Studien, no 211, September 1992, pp. 140-161.

[KLS92] M. Kramer, G. Lausen, and G. Saake: Updates in a Rule-Based Language for Objects. In Manuskripte, Reihe Informatik, no 1/1992, Fakultät für Mathematik und Informatik, University of Mannheim, 1992.

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

[Kor87] J. Kornell: Formal Thought and Narrative Thought in Knowledge Acquisition, *International Journal on Man-Machine Studies*, vol 26, no 2, 1987, pp. 203-212.

[KoT87] Y. Kodratoff and G. Tecuci: Techniques of Design and DISCIPLE Learning Apprentive, *International Journal of Expert Systems*, vol 1, 1987, pp. 39-66.

[KoT90] W. Kowalczyk and J. Treur: On the Use of a Formalized Generic Task Model in Knowledge Acquisition. In B. Wielinga et al. (eds.)*, Current Trends in Knowledge Acquisition*, IOS Press, Amsterdam, 1990, pp. 198-221.

[Kow74] R. A. Kowalski: Predicate Logic as a Programming Language. In *Proceedings of the Information Processing (IFIP Congress 74)*, Stockholm, Sweden, August 5-10, 1974, pp. 569-574.

[Koz90] D. Kozen: Logics of Programs. In J. v. Leeuwen (ed.), *Handbook of Theoretical Computer Science*, Elsevier Science Publ., B. V., Amsterdam, 1990, pp. 789-840.

[Kra87] B. Krämer: SEGRAS - A Formal and Semigraphical Language Combining Petri Nets and Abstract Data Types for the Specification of Distributed Systems. In *Proceedings of the 9th International Conference on Software Engineering*, March 30th - April 2th, Monterey, CA, 1987, pp. 116-125.

[KrR89] R. Krista and I. Rozman: A Computer Aided Prototyping Methodology, *ACM SIGSOFT Software Engineering Notes*, no 6, vol 14, October 1989, pp. 68-72.

[KRT89] S. Katz, C.A. Richter, and K. The: PARIS: A System for Reusing Partially Interpreted Schemas. In T. J. Biggerstaff et al (eds.), *Frontier Series: Software Reusability: Volume I - Concepts and Models,* ACM Press, New York, 1989, pp. 257-273.

[Kru92] C. W. Krueger: Software Reuse, *ACM Computing Surveys*, vol 24, no 2, June 1992, pp. 131-184.

[Kun89] C. H. Kung: Conceptual Modeling in the Context of Software Development, *IEEE Transaction on Software Engineering*, vol 15, no 10, 1989, pp. 1176-1187.

[KVS91] W. Karbach, A. Voß, R. Schuckey, and U. Drouven: MODEL-K: Prototyping at the Knowledge Level. In *Proceedings of the 11th International Conference on Expert Systems and their Applications, vol 1, General Conference Tools, Techniques and Methods*, May 27-31, Avignon, 1991, pp 501-511.

[Lan93] D. Landes: Entwicklung wissensbasierter Systeme auf der Grundlage einer ausführbaren Spezifikation. In *Proceedings of Expertensysteme ´93*, Hamburg, February 17-19, 1993 (in German). Also appeared as: Developing Knowledge-Based Systems on the Basis of an Executable Specification, Institut für Angewandte Informatik und Formale Beschreibungsverfahren, University of Karlsruhe, research report, no 265, 1993.

[LDG91] N. Lavrac, S. Dzeraski, and M. Grobelnik: Learning of Nonrecursive Definitions of Relations with Linus. In Y. Kodratoff (ed.), *Proceedings of the European Working Session on Learning (EWSL)´*91, Porto, Portugal, March 6-8, 1991, Lecture Notes in Artificial Intelligence, no 482, Springer-Verlag, Berlin, 1991, pp. 265-281.

[Lee85] S. Lee: On Executable Models for Rule-Based Prototyping. In *Proceedings of the 8th International Conference on Software Engineering*, London, UK, August 28-30, 1985, pp. 210-215.

[LeS84] U. Leibrandt and P. Schnupp: An Evaluation of Prolog as a Prototyping System. In Budde et al. (eds.), *Approaches to Prototyping*, Springer-Verlag, Berlin, 1984, pp. 425-433.

[LFA93] D. Landes, D. Fensel, and J. Angele: Formalizing and Operationalizing a Design Task with KARL. In J. Treur et al. (eds.), *Formal Specification of Complex Reasoning Systems*, Proceedings of the ECAI-92 Workshop on Formal Specification Methods for Complex Reasoning Systems, European Conference on Artificial Intelligence (ECAI-92), Wien, Austria, August 3-7, 1992, Ellis Horwood, Chichester, to appear 1993.

[LFF91] M. Leszak, B. Franzen, and K. Fritschi: Integration Strukturierter Methoden zur Anforderungsspezifikation von Informationssystemen (Integrating Structured Methods for Requirements Engineering of Information Systems). In *Proceeedings of Requirements Engineering´91,* April 10-11, 1991, Marburg, Informatik-Fachberichte, no 273, Springer-Verlag, Berlin, 1991, pp. 139-159 (in German).

[LHS92] D. Landes, D. Hackenberg, and T. Schweier: An Inference Structure for a Configuration Problem. 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.

[LiM92] M. Linster and M. Musen: The Inference Structure of K-ONCOCIN: Skeletal Plan Refinement. 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.

[Lin91] M. Linster: Tackling the Office-Plan Problem with OMOS. In [Lin92c].

[Lin92a] M. Linster: Knowledge Acquisition Based on Explicit Methods of Problem Solving, PhD thesis, University of Kaiserslautern, February 1992.

[Lin92b] M. Linster: Linking Modeling to Make Sense and Modeling to Implement Systems in an Operational Modeling Environment. In *Proceedings of the 6th European Knowledge Acquisition for Knowledge-Based Systems Workshop (EKAW-92)*, May 18-22, Heidelberg/Kaiserslautern, 1992, T. Wetter et al. (eds.), *Current Developments in Knowledge Acquisition*, Lecture Notes in Artificial Intelligence, no 599, Springer-Verlag, Berlin, 1992, pp. 55-74.

[Lin92c] M. Linster (ed.): Sisyphus ´91: Models of Problem Solving, Arbeitspapiere der GMD, no 630, March 1992.

[Lin92d] M. Linster (ed.): Sisyphus ´92: Models of Problem Solving, Arbeitspapiere der GMD, no 663, July 1992.

[Lin92e] M. Linster: Using the Operational Modelling Language OMOS to Tackle the Sisyphus´92 Office-Planning Problem. In [Lin92d].

[LKV92] M. Linster, W. Karbach, A. Voß, and J. Walther: An Analysis of the Role of Operational Modelling Languages in the Development of Knowledge-Based Systems, *Proceedings of the 2nd Japanese Knowledge Acquisition for Knowledge-Based Systems Workshop (JKAW´1992)*, Hatoyama, Japan, November 9-13, 1992, pp. 65-88.

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

[LMW90] L. Logrippo, T. Melanchuk, and R. J. D. Wors: The Algebraic Specification Language LOTOS: An Industrial Experience. In *Proceedings of the ACM SIGSOFT International Workshop on Formal Methods in Software Development*, Napa, California, May 9-11, 1990, pp. 59-66.

[LNO89] G. Lausen, T. Nemeth, A. Oberweis, F. Schönthaler, and W. Stucky: The INCOME Approach for Conceptual Modelling and Prototyping of Information Systems. In *Proceedings of the 1st Nordic Conference on Advanced Systems Engineering CASE'89*, Stockholm, Sweden, May 9-11, 1989.

[LNP87] K. von Luck, B. Nebel, C. Peltason, and A. Schmiedel: The Anatomy of the BACK-System. KIT-BACK report, no 41, Technische Universität Berlin, January 1987.

[LPT93] Izak van Langevelde, A. Philipsen, and J. Treur: A Compositional Architecture for Simple Design Formally Specified in DESIRE. In J. Treur et al. (eds.), *Formal Specification of Complex Reasoning Systems*, Proceedings of the ECAI-92 Workshop on Formal Specification Methods for Complex Reasoning Systems, European Conference on Artificial Intelligence (ECAI-92), Wien, Austria, August 3-7, 1992, Ellis Horwood, Chichester, 1993.

[Mai86] D. Maier: A Logic for Objects. In *Proceedings of the Workshop on Foundations of Deductive Databases and Logic Programming*, Washington D.C., August 1986, pp. 6-26.

[MaM85] L. Mathiassen and A. Munk-Madsen: Formalization in System Development, H. Ehrig et al. (eds.), *Formal Methods and Software Development, vol 2*, Lecture Notes in Computer Science, vol 186, Springer-Verlag, Berlin, 1985, pp. 101-116.

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

[McD88] J. McDermott: Preliminary Steps Toward a Taxonomy of Problem-Solving Methods. In [Mar88], pp. 225-255.

[McH89] K. L. McGraw and K. Harbison-Briggs: *Knowledge Acquisition. Principles and Guidelines*, Prentice-Hall International, Englewood Cliffs, NJ, 1989.

[MDK92] D. Marques, G. Dallamagne, G. Klinker, J. McDermott, and D. Tung: Easy Programming. Emporing

People to Build Their Own Applications, *IEEE Expert*, vol 7, no 3, 1992, pp. 16-29.

[MeB91] M. Meyer and J. Booker: *Eliciting and Analyzing Expert Judgement. A Practical Guide*, Knowledge-Based Systems, vol 5, Academic Press, Londen, 1991.

[MMH86] R. S. Michalski, I. Mozetic, J. Hong and N. Lavrac: The Multi-Purpose Incremental Learning System AQ15 and its Testing Application to Three Medical Domains. In *Proceedings of the 5th National Conference on AI (AAAI-86)*, Philadelphia, PA, August 11-15, pp. 1041-1045, 1986.

[Möl92] J.-U. Möller: Towards Declarative Programming in Conceptual Models. In *Proceedings of the 2nd Workshop Informationssysteme und Künstliche Intelligenz*, February 24-26, Ulm, 1992, R. Studer (ed.), Informatik-Fachberichte, no. 303, Springer-Verlag, Berlin, 1992.

[Mor87a] K. Morik: Sloppy Modeling. In K. Morik (ed.), *Knowledge Representation and Organisation in Machine Learning*, Lecture Notes in Artificial Intelligence, no 347, Springer-Verlag, Berlin, 1987, pp. 107-134.

[Mor87b] K. Morik: Acquiring Domain Models, *International Journal of Man-Machine Studies*, vol 26, 1987, pp. 93-104.

[Mos83] M. G. Moser: An Overview of NIKL, the New Implementation of KL-ONE, Research in Knowledge Representation for Natural Language Understanding, Bolt Beranek Newman, Annual Report, no 5421, 1983.

[MRD91] E. Motta, T. Rajan, J. Domingue, and M. Eisenstadt: Methodological Foundations of KEATS, the Knowledge Engineer´s Assistant, *Knowledge Acquisition*, vol 3, no 1, March 1991, pp. 21-47.

[Mug92] S, Muggleton (ed.): *Inductive Logic Programming*, Academic Press, London, 1992.

[Mus89a] M. A. Musen: Conceptual Models of Interactive Knowledge Acquisition Tools, *Knowledge Acquisition*, vol 1, no 1, March 1989, pp. 73-98.

[Mus89b] M. A. Musen: *Automated Generation of Model-Based Knowledge-Acquisition Tools*, Morgan Kaufmann Publisher, San Mateo, CA, 1989.

[Mus92] M. Musen: Overcoming the Limitations of Role-Limiting Methods, *Knowledge Acquisition*, vol 4, no 2, June 1992, pp. 165-170.

[NaS73] I. Nassi and B. Shneiderman: Flowchart Techniques for Structured Programming, *ACM SIGPLAN Notices*, vol 8, no 8, 1973, pp. 12-16.

[Nau82] P. Naur: Formalization in Program Development, *BIT*, vol 22, 1982, pp. 437-453.

[Nau85] P. Naur: Programming as Theory Building, *Microprocessing and Microprogramming*, vol 15, 1985, pp. 253-261.

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

[NeS76] A. Newell and H. A. Simon: Computer Science as Empirical Inquiry: Symbols and Search, *Communications of the ACM*, vol 19, no 3, March 1976, pp. 113-126.

[NeS92] S. Neubert and R. Studer: The KEEP Model. In *Proceedings of the 6th European Knowledge Acquisition for Knowledge-Based Systems Workshop (EKAW-92)*, May 18-22, Heidelberg/ Kaiserslautern, 1992, T. Wetter et al. (eds.), *Current Developments in Knowledge Acquisition*, Lecture Notes in Artificial Intelligence, no 599, Springer-Verlag, Berlin, 1992, pp. 230-249.

[New80] A. Newell: Physical Symbol Systems, *Cognitive Science*, vol. 4, 1980, pp. 135-183.

[New82] A. Newell: The Knowledge Level, *Artificial Intelligence*, vol 18, 1982, pp. 87-127.

[NiM90] U. Nilson and J. Matuszynski: *Logic, Programming and Prolog*, John Wiley and Sons, Chichster, 1990.

[NOS92] T. Nemeth, A. Oberweis, F. Schönthaler, and W. Stucky: INCOME: Arbeitsplatz für den Programmentwurf interaktiver Informationssysteme (INCOME: Working Place for the Development of Interactive Information Systems), research report, Institut für Angewandte Informatik und Formale Beschreibungsverfahren, University of Karlsruhe, no 251, August 1992 (in

German).

[Obe90] A. Oberweis: Petri-Netz-Beschreibungstechniken für Exception-Handling: Mechanismen in der Automatisierungstechnik (Petri-Nets as Representation Formalism for Exception-Handling), research report, Institut für Angewandte Informatik und Formale Beschreibungsverfahren, University of Karlsruhe, no 207, 1990 (in German).

[Ohl92] C. Ohlgart: Spezifikation eines wissensbasierten Systems zur Konfiguration eines Versicherungsschutzes für Privatkunden (Specifikation of a Knowledge-Based System for Configuration of Insurance Packages), master thesis, Institut für Angewandte Informatik und Formale Beschreibungsverfahren, University of Karlsruhe, 1992 (in German).

[OlR87] J. R. Olson and H. H. Rueter: Extracting Expertise from Experts: Methods for Knowledge Acquisition, *Expert Systems*, vol 4, no 3, 1987, pp. 152-169.

[Par86] D. Partridge: *Aritificial Intelligence. Application In The Future Of Software Engineering*, Ellis Horwood Limited, Great Britain, 1986.

[PET92] A. R. Puerta , J. W. Edgar, S. W. Tu, and M. A. Musen: A Multiple-Method Knowledge-Acquisition Shell for the Automatic Generation of Knowledge-Acquisition Tools, *Knowledge Acquisition*, vol 4, no 2, pp. 171-196, 1992.

[Pre87] R. S. Pressman: *Software Engineering. A Practitioner´s Approach*, 2nd edition, McGraw-Hill Book Comp., New York, 1987.

[PrL89] N. S. Prywes and E. D. Lock: Use of the MODEL Equation Language and Program Generator By Management Professionals. In T. J. Biggerstaff et al. (eds.), *Frontier Series: Software Reusability - vol I - Concepts and Models*, ACM Press, New York, 1989, pp. 103-129.

[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, pp. 192-216.

[PSF91] G. Piatetsky-Shapiro and W. J. Frawley: *Knowledge Discovery in Databases*, AAAI Press / The MIT Press, Menlo Park, California, 1991.

[PuG92] F. Puppe and U. Gappa: Towards Knowledge Acquisition by Experts. In F. Belli et al. (eds.), *Industrial and Engineering Applications of Artifical Intelligence and Expert Systems, Proceedings of the 5th International Conference IEA/AIE-92*, Paderborn, Lecture Notes in Artificial Intelligence, no 604, Springer-Verlag, Berlin, pp. 546-555, 1992.

[Qui84] J.R. Quinlan: Learning Efficient Classification Procedures and their Application to Chess End Games. In R.S. Michalski et al. (eds.), *Machine Learning. An Artificial Intelligence Approach, vol 1,* Springer-Verlag, Berlin, 1984, pp. 463-482.

[Qui90] J. R. Quinlan: Learning Logical Definitions from Relations, *Machine Learning*, vol 5, no 3, 1990, pp. 239-266.

[RaP92] P. Rademakers and R. Pfeifer: The Role of Knowledge Level Models in Situated Adaptive Design, *Proceedings of the 10th European Conference on Artificial Intelligence (ECAI´92)*, August 3-7, Wien, Austria, 1992, pp. 601-602.

[Rei78] R. Reiter: On Closed World Data Bases. In H. Gallaire et al. (eds.), *Logic and Data Bases*, Plenum Press, New York, 1978, pp. 55-76.

[Rei86] W. Reisig: Place/Transition Systems. In W. Brauer et al. (eds.), *Petri Nets: Central Models and Their Properties, Advances in Petri Nets, 1986, Part I, Proceedings of an Advanced Course*, Bad Honnef, September 8-19th, 1986, Lecture Notes in Computer Science, no 254, Springer-Verlag, Berlin, 1986, pp. 117-141.

[Rei90] W. Reisig: *Petrinetze. Eine Einführung* (Petri Nets. An Introduction), second edition, Springer-Verlag, Berlin, 1990 (in German).

[Rob65] J. A. Robinson. A Machine-Oriented Logic Based on the Resolution Principle, *Journal of the Association for Computer Machinery*, vol 12, no 1, January 1965, p. 23-41.

[Ros77] D.T. Ross: Structured Analysis (SA): a Language for Communicating Ideas, *IEEE Transactions on*

Software Engineering, vol 3, no 1, January 1977, pp. 16 - 34.

[Roy70] W. W. Royce: Managing the Development of Large Software Systems: Concepts and Techniques. In *Proceedings WESCON*, August 1970.

[San92] P. Sander: Eine ordnungsbasierte Regelsprache für NF2-Relationen (A Rule Language for NF2-Relations), PhD thesis, University of Karlsruhe, 1992 (in German).

[SAW90] G. Schreiber, H. Akkermans, and B. Wielinga: On Problems with the Knowledge Level Perspective. In *Proceedings of the 5th Knowledge Acquisition for Knowledge-Based Systems Workshop*, Banff, Canada, November 4-9, 1990.

[Say90] H. H. Sayani: PSL/PSA at the Age of Fifteen. In R. H. Thayer et al. (eds.), *System and Software Requirements Engineering*, IEEE Computer Society Press, Washington, 1990, pp. 403-417.

[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.

[Sch92a] G. Schreiber: Pragmatics of the Knowledge Level, PhD Thesis, University of Amsterdam, 1992.

[Sch92b] P. H. Schmitt: *Theorie der logischen Programmierung* (Theory of Logic Programming), Springer-Verlag, Berlin, 1992 (in German).

[SDD86] J. T. Schwarz, R. Dewar, E. Dubinski, and E. Schonberg: *Programming with Sets: An Introduction to SETL*, Springer-Verlag, Berlin, 1986.

[ShD90] J. W. Shavlik and T. G. Dietterich (eds.): *Readings in Machine Learning*, Morgan Kaufmann Publisher, San Mateo, California, 1990.

[She88] J. C. Shepherdson: Negation in Logic Programming. In J. Minker (ed.), *Foundations of Deductive Databases and Logic Programming*, Morgan Kaufmann Publisher, Los Altos, CA, 1988, pp.19-88.

[Shu89] R. N. Shutt: A Rigorous Development Strategy Using the OBJ Specification Language and the MALPAS Program Analysis Tool. In *Proceedings of the 2nd European Software Engineering Conference ESEC´89*, Warwick, UK, September 11-15, 1989, Lecture Notes in Computer Science, no 387, Springer-Verlag, Berlin, 1989, pp. 260-291.

[Smi90] D. R. Smith: KIDS: A Semiautomatical Program Development System, *IEEE Transaction on Software Engineering*, vol 16, no 9, September 1990, pp. 1024-1043.

[Som89] I. Sommerville: *Software Engineering*, 3rd edition, Addison-Wesley, Workingham, England, 1989.

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

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

[Spi89] J. M. Spivey: *The Z Notation: A Reference Manual*, Prentice-Hall, Englewood Cliffs, 1989.

[Ste92] L. Steels: Reusability and Configuration of Applications by Non-Programmers. VUB AI-Memo 92-4, University of Bruessel, April, 1992.

[StS88] J. Stelovsky and H. Sugaya: A System for Specification and Rapid Prototyping of Application Command Languages, *IEEE Transactions on Software Engineering*, no 7, vol 14, July 1988, pp. 1023- 1032.

[Stu87] R. Studer: *Konzepte für eine verteilte wissensbasierte Softwareproduktionsumgebung* (Concepts for Distributed Knowledge-Based Software Development Environments). In Informatik-Fachberichte, no 132, Springer-Verlag, Berlin, 1987 (in German).

[Sum77] P. D. Summer: A Methodology for LISP Program Construction from Examples, *Journal of the Association for Computing Machinery*, vol 24, no 1, January 1977, pp. 161-175.

[SWA92] G. Schreiber, B. Wielinga, and H. Akkermans: Differentiating Problem Solving Methods. In *Proceedings of the 6th European Knowledge Acquisition for Knowledge-Based Systems Workshop (EKAW-92)*, May 18-22, Heidelberg/Kaiserslautern, 1992. T. Wetter et al. (eds.), Current Developments in Knowledge Acquisition, Lecture Notes in Artificial Intelligence, no 599, Springer-

Verlag, Berlin, 1992, pp. 95-111.

[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, San Diego, 1993.

[Tav85] R. D. Tavendale: A Technique for Prototyping Directly From a Specification. In *Proceedings of the 8th International Conference on Software Engineering*, London, UK, August 28-30, 1985 , pp. 224-229.

[TeH77] D. Teichroew and E. A. Hershey: PSL/PSA: a Computer-Aided Technique for Structured Documentation and Analysis of Information Processing Systems, *IEEE Transactions on Software Engineering*, vol 3, no 1, January 1977, pp. 41-48.

[ThD90] R. H. Thayer and M. Dorfman (eds.): *System and Software Requierements Engineering*, IEEE Computer Society Press, Los Alamitos, California, 1990.

[THR91] A. t. Teije, F.v. Harmelen, and M. Reinders: Si(ML)^2: a Prototype Interpreter for a Subset of (ML)^2, ESPRIT project P5248 KADS-II, report KADS-II/T1.2/TR/UvA/005/1.0, University of Amsterdam, 1991.

[Tra88a] W. Tracz: Software Reuse Myths, *ACM Sigsoft Software Engineering Notes*, vol 13, no 1, January 1988, pp. 17-21.

[Tra88b] W. Tracz: Software Reuse Maxims, *ACM Sigsoft Software Engineering Notes*, vol 13, no 4, October 1988, pp. 28-31.

[TWJ92] J. J. P. Tsai, T. Weigert, and H.-C. J. Jang: A Hybrid Knowledge Representation as a Basis of Requirements Specification and Specification Analysis, *IEEE Transactions on Software Engineering*, vol 18, no 12, December 1992.

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

[Ull89] J. D. Ullman: *Principles of Database and Knowledge-Base Systems, vol II*, Computer Sciences Press, Rockville, Maryland, 1989.

[Ven84] R. Venken: Prolog as a Language for Prototyping of Information Systems. In Budde et al. (eds.), *Approaches to Prototyping*, Springer-Verlag, Berlin, 1984, pp. 447-458.

[Vin92] R. E. Vinkhuyzen: On the Non-Existence of Knowledge Level Models, *Proceedings of the 10th European Conference on Artificial Intelligence (ECAI´92)*, August 3-7, Vienna, Austria, 1992, pp. 620-622.

[VJS93] L. in ´t Veld, Willem Jonker, and J. W. Spee: The Specification of Complex Reasoning Tasks in K_{BS}SF. In J. Treur et al. (eds.), *Formal Specification of Complex Reasoning Systems*, Proceedings of the ECAI-92 Workshop on Formal Specification Methods for Complex Reasoning Systems, European Conference on Artificial Intelligence (ECAI-92), Wien, Austria, August 3-7, 1992, Ellis Horwood, Chichester, to appear 1993.

[VKS91] A. Voß, W. Karbach, R. Schuckey, and U. Drouven: The Office Planning Problem in MODEL-K. In [Lin92c].

[VoK93] A. Voß and W. Karbach: MODEL-K: Making KADS Run. In G. Schreiber et al. (eds.), *KADS: Knowledge Acquisition and Design Structuring*, Academic Press, London, forthcoming.

[VVW93] A. Voß, H. Voß, J. Walther, and T. Hemman: Model-Driven Prototyping - Prototyping-Driven Modelling in Knowledge-Based Systems, *Proceedings of Requirements Engineering ´93 - Prototyping -*, Bonn, April 25 - 27, 1993, Teubner Verlag, Stuttgart, to appear 1993.

[Wan88] Y. Wang: A Distributed Specification Model and Its Prototyping, *IEEE Transactions on Software Engineering*, no 8, vol 14, August 1988, pp. 1090-1097.

[WeS91] T. Wetter and W. Schmidt: Formalization of the KADS Interpretation Models. In Proceedings of the 8th Conference of the Society for the Study of Artificial Intelligence and Simulation of Behaviour (AISB91), Leeds, April 16-19, 1991, pp. 172-183.

[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, pp. 356-375.

[Wet92] T. Wetter: FORKADS: An Executable Language for the KADS Conceptual and Interpretation Models, Habilitationsschrift, University of Kaiserslautern, Germany, 1992.

[Wir88] N. Wirth: *Programming in Modula-2, 4th ed.*, Springer-Verlag, Berlin, 1988.

[WSB92] B.J. Wielinga, A.Th. Schreiber, and J.A. Breuker: KADS: A Modelling Approach to Knowledge Engineering, *Knowledge Acquisition*, vol 4, no 1, March 1992, pp. 5-53.

[WVL92] J. Walther, A. Voß, M. Linster, T. Hemman, H. Voß, and W. Karbach: MoMo, GMD-Arbeitspapiere, no 658, Juni 1992.

[Yag92] R. R. Yager (ed.): Special Issue Knowledge Discovery in Data- and Knowledge Bases, *International Journal of Intelligent Systems*, vol 7, no 7, September 1992.

[YKS85] H. Yoshida, H. Kato, and M. Sugimoto: Retrieval of Software Module Functions Using First-Order Predicate Logical Formulae. In *Proceedings of the 4th Conference Logic Programming´85*, Tokio, Japan, July 1-3, 1985, pp. 117-127.

[YoC79] E. Yourdon and L. L. Constantine: *Structured Design: Fundamentals of a Discipline of Computer Program and System Design*, Yourdon Press, Englewood Cliffs, NJ, 1979.

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

[Zav82] P. Zave: An Operational Approach to Requirements Specification for Embedded Systems, *IEEE Transactions on Software Engineering*, no 3, vol 8, May 1982, pp. 250-269.

[Zav91] P. Zave: An Insider´s Evaluation of PAISLey, *IEEE Transactions on Software Engineering*, vol 17, no 3, 1991, pp. 212-225.

List of Definitions

List of Figures

List of Symbols

A	a semantical structure of a P-KARL language P.
$A_{E\rightarrow}$	a finite set of single-valued attributes for elements.
$A_{E\rightarrow\rightarrow}$	a finite set of set-valued attributes for elements.
A_{Π}	a finite set of argument names for predicates.
$A_{\Sigma\rightarrow}$	a finite set of single-valued attributes for classes.
$A_{\Sigma\rightarrow\rightarrow}$	a finite set of set-valued attributes for classes.
B	a finite set of boolean variables $\{b_1, b_2, ..., b_m\}$ of a P-KARL language.
CD(H)	denotes the set of all terms from H over CD.
cia_j	denotes a composed inference action.
^{cai_j}InferenceActions	is a set of names of elementary and composed inference actions and defines the internal inference actions of the composed inference action cia_j.
^{cia_j}InputStores	is a set of role names and defines the input roles, i.e. the premises, of the composed inference action cia_j which are stores.
^{cia_j}InternalStores	is a set of role names and defines the internal roles of the composed inference action cia_j which are stores.
^{cai_j}InternalTerminators	is a set of role names and defines the internal roles of the composed inference action cia_j which are terminators.
^{cia_j}InternalViews	is a set of role names and defines the internal roles of the composed inference action cia_j which are views.
^{cia_j}OutputStores	is a set of role names and defines the output roles, i.e. the premises, of the composed inference action cia_j which are stores.
^{cia_j}Terminators	is a set of role names and defines the output roles, i.e. the conclusions, of the composed inference action cia_j which are terminators.
^{cia_j}Views	is a set of role names and defines the input roles, i.e. the premises, of the composed inference action cia_j which are views.
CIA	is a finite set of names which defines the names of the composed inference actions of an inference layer.
D	is a set which is used to interpret a P-KARL language P.

${}^{D}A_{\Pi}$	defines the domain-specific argument names for predicates.
${}^{D}A_{E\rightarrow}$	defines the domain-specific single-valued attributes for elements.
${}^{D}A_{E\rightarrow\rightarrow}$	defines the domain-specific set-valued attributes for elements.
${}^{D}A_{\Sigma\rightarrow}$	defines the domain-specific single-valued attributes for classes.
${}^{D}A_{\Sigma\rightarrow\rightarrow}$	defines the domain-specific set-valued attributes for classes.
${}^{D}C$	$= ({}^{D}C_1, {}^{D}C_2)$ is a set of constraints, i.e. necessary conditions ensuring consistency of a domain layer.
${}^{D}D$	a set of definitions over a domain layer alphabet ${}^{D}L$.
${}^{D}E$	defines the domain-specific element denotations.
${}^{D}F$	a set of facts over a domain layer alphabet ${}^{D}L$.
$DG_L(S')$	the dependency graph of an L-KARL language L set of rules S´.
${}^{D}L$	defines the domain-specific L-KARL alphabet.
${}^{D}R$	a set of rules over a domain layer alphabet ${}^{D}L$.
${}^{D}\Sigma$	defines the domain-specific class names.
${}^{D}\Pi$	defines the domain-specific predicate symbols.
${}^{D\cup I}L$	the union of the alphabet of the domain layer ${}^{D}L$ and the alphabet of the inference layer ${}^{I}L$.
E	a finite set of element constructors. The elements of “E” are function symbols with an arity $0 \le k$. 0-ary symbols in “E” are called constants.
eia_j	denotes an elementary inference action.
${}^{eia_j}C$	is a pair of two possibly empty set of constraints $({}^{eia_j}C_1, {}^{eia_j}C_2)$ of the elementary inference action eia_j.
${}^{eia_j}D$	is a set of class definitions and predicate definitions of the elementary inference action eia_j.
${}^{eia_j}\mathrm{InputStores}$	is a possible empty set of store names and defines input stores, i.e. premises, of the elementary inference action eia_j.
${}^{eia_j}\mathrm{OutputStores}$	is a possible empty set of store names and defines output stores, i.e. conclusions, of the elementary inference action eia_j.
${}^{eia_j}R$	is a set of rules of the elementary inference action eia_j.
${}^{eia_j}\mathrm{Terminators}$	is a set of terminators names and defines views which are conclusions of the elementary inference action eia_j.

${}^{eia_j}Views$	is a set of view names and defines views which are premises of the elementary inference action eia_j.
EIA	is a finite set of names which defines the names of the elementary inference actions of an inference layer.
F	a finite set $\{f_1, f_2, ..., f_r\}$ of k-ary (n > 0) function symbols of a P-KARL language.
F^A	is a set $\{f_1{}^A, f_2{}^A, ..., f_r{}^A\}$ which is used to interpret every function symbol $f_i \in F$ of a P-KARL language P by a n-ary function f^A.
H	a Herband model.
H´	an untyped Herbrand model.
HB	a Herbrand basis.
HI	a Herbrand interpretation.
HI´	an untyped Herbrand interpretation.
HU	a Herbrand universum.
i	$= (\|i\|, {}^iD)$ is a set of facts and class definitions representing case-specific input data at a domain layer.
I	a semantical structure of an L-KARL language.
${}^IA_{E\rightarrow}$	defines the generic single-valued attributes for elements at the inference layer.
${}^IA_{\Sigma\rightarrow}$	defines the generic single-valued attributes for classes at the inference layer.
${}^IA_{E\rightarrow\rightarrow}$	defines the generic set-valued attributes for elements at the inference layer.
${}^IA_{\Sigma\rightarrow\rightarrow}$	defines the generic set-valued attributes for classes at the inference layer.
${}^IA_{\Pi}$	defines the generic argument names for predicates at the inference layer.
$I_{A\Pi}$	interprets each k-ary predicate symbol $p \in \Pi$ as a k-ary tuple $(T_1, ..., T_k)$ with $T_i \in \wp\uparrow_{\leq_U} (U_\Sigma \cup \{U_{VB}, U_{VI}, U_{VR}, U_{VS}\})$ for all i = 1,...,k.
${}^iD$	definitions of the input-data part of a domain layer.
ID*	$:= E^* \cup \Sigma \cup V$.
${}^IE$	defines the generic element denotations at the inference layer.

$I_{E\rightarrow}$	interprets each single-valued attribute $a \in A_{E\rightarrow}$ as a partial function $I_{E\rightarrow}(a): U_E \rightarrow U$
$I_{E\rightarrow\rightarrow}$	interprets each set-valued attribute $s \in A_{E\rightarrow\rightarrow}$ as a partial function $I_{E\rightarrow\rightarrow}(s): U_E \rightarrow \wp(U)$
$I_{E\Rightarrow}$	interprets each single-valued attribute $a \in A_{E\rightarrow}$ as a partial anti-monotonic function vs. "$\leq_U$" and "$\subseteq$" having the upwards-closed subsets of $\wp\uparrow_{\leq_U}(A)$ as range with $A = U_\Sigma \cup \{U_{VB}, U_{VI}, U_{VR}, U_{VS}\}$. $I_{E\Rightarrow}(a): U_E \cup U_\Sigma \rightarrow \wp\uparrow_{\leq_U}(U_\Sigma \cup \{U_{VB}, U_{VI}, U_{VR}, U_{VS}\})$.
$I_{E\Rightarrow\Rightarrow}$	interprets each set-valued attribute $s \in A_{E\rightarrow\rightarrow}$ as a partial anti-monotonic function vs. "$\leq_U$" and "$\subseteq$" having the upwards-closed subsets of $\wp\uparrow_{\leq_U}(A)$ as range with $A = U_\Sigma \cup \{U_{VB}, U_{VI}, U_{VR}, U_{VS}\}$ $I_{E\Rightarrow\Rightarrow}(s): U_E \cup U_\Sigma \rightarrow \wp\uparrow_{\leq_U}(U_\Sigma \cup \{U_{VB}, U_{VI}, U_{VR}, U_{VS}\})$.
${}^I L$	defines a generic L-KARL alphabet used at the inference layer.
$\overline{Input}_{eia_j}$	defines an ***input*** of an elementary inference action eai_j.
${}^I S$	is a finite set which defines the store names of an inference layer.
IS	$= (IS_V, IS_E)$ denotes an inference structure.
IS_{cia_j}	$= (IS_{V_{cia_j}}, IS_{E_{cia_j}})$ denotes a substructure of an inference structure IS according to a composed inference action cia_j.
IS_E	denotes the edges of an inference structure IS.
$IS_{E_{cia_j}}$	denotes the edges of a substructure of an inference structure IS_{cia_j}.
IS_V	denotes the vertices of an inference structure IS.
$IS_{V_{cia_j}}$	denotes the vertices of a substructure of an inference structure IS_{cia_j}.
${}^I T$	is a finite set which defines the terminator names of an inference layer.
I_U	interprets every element of $E \cup \Sigma \cup V$.
${}^I V$	is a finite set which defines the view names of an inference layer.
${}^I \Pi$	defines the generic predicate symbols at the inference layer.
I_Π	interprets each k-ary predicate symbol $p \in \Pi$ by a relation $I_\Pi(p) \subseteq U^k$.
${}^I \Sigma$	defines the generic class names at the inference layer.
$I_{\Sigma\rightarrow}$	interprets each single-valued attribute $a \in A_{\Sigma\rightarrow}$ as a partial function $I_{\rightarrow}(a): U_\Sigma \rightarrow U$
$I_{\Sigma\rightarrow\rightarrow}$	interprets each set-valued attribute $s \in A_{\Sigma\rightarrow\rightarrow}$ as a partial function $I_{\Sigma\rightarrow\rightarrow}(s): U_\Sigma \rightarrow \wp(U)$

$I_{\Sigma\Rightarrow}$	interprets each single-valued attribute $a \in A_{\Sigma\rightarrow}$ as a partial anti-monotonic function vs. "$\leq_U$" and "$\subseteq$" having the upwards-closed subsets of $\wp_{\uparrow_{\leq_U}}(A)$ as range with $A = U_\Sigma \cup \{U_{VB}, U_{VI}, U_{VR}, U_{VS}\}$. $I_{\Sigma\Rightarrow}(a)\colon U_\Sigma \rightarrow \wp_{\uparrow_{\leq_U}}(U_\Sigma \cup \{U_{VB}, U_{VI}, U_{VR}, U_{VS}\})$.
$I_{\Sigma\Rightarrow\Rightarrow}$	interprets each set-valued attribute $s \in A_{\Sigma\rightarrow\rightarrow}$ as a partial anti-monotonic function vs. "$\leq_U$" and "$\subseteq$" having the upwards-closed subsets of $\wp_{\uparrow_{\leq_U}}(A)$ as range with $A = U_\Sigma \cup \{U_{VB}, U_{VI}, U_{VR}, U_{VS}$ $I_{\Sigma\Rightarrow\Rightarrow}(s)\colon U_\Sigma \rightarrow \wp_{\uparrow_{\leq_U}}(U_\Sigma \cup \{U_{VB}, U_{VI}, U_{VR}, U_{VS}\})$.
l_i	a literal of an L-KARL language.
L	an L-KARL language.
$N_L(S')$	all ground atoms constructable in an L-KARL language L.
o	the dataflow variable of a P-KARL languge which refers to the output-data part of a domain layer. Or: $o = (\|o\|, {}^oD)$ is a set of facts and class definitions representing case-specific output data at a domain layer.
${}^oD$	definitions of the output-data part of a domain layer.
p	a P-KARL program.
p^A	the interpretation of a P-KARL program p.
P	a P-KARL language.
PS	a set of predicate symbols of a P-KARL language.
$Rule_{eia_j}$	$:= {}^{eia_j}R \cup Vrule_{eia_j} \cup Trule_{eia_j}$ defines the ***rule set*** of the elementary inference action eia_j.
S	a set of L-KARL clauses and definitions.
S^A	the set of all variable assignments of a P-KARL language P and its interpretation A.
$store_i$	denotes a store.
${}^{store_i}D$	is a set of class definitions over the alphabet ${}^IL$. It is defined in a store.
S´	a set of rules.
S´*	denotes the set of all ground instances of a rule set S´ over an L-KARL language L.
$terminator_i$	denotes a terminator.

${}^{terminator_i}D$	is a set of class definitions and predicate definitions over the alphabet ${}^{I}L$. It is defined in a terminator.
${}^{terminator_i}R$	is a set of rules. It is defined in a terminator.
$Term_{eia_j}$	$:= {}^{eia_j}D \cup Iterm_{eia_j} \cup Vterm_{eia_j} \cup Tterm_{eia_j} \cup Oterm_{eia_j}$ defines the terminology of the elementary inference action eia_j.
U	$:= U_E \cup U_\Sigma \cup U_V$.
U_E	is a non-empty set of all elements.
U_V	$:= U_{VB} \cup U_{VI} \cup U_{VR} \cup U_{VR}$.
U_{VB}	:= {False, True}
U_{VI}	is the set of all integers.
U_{VR}	is the set of all real numbers.
U_{VS}	is the set of all strings of finite length.
U_Σ	is a non-empty set of all classes.
V	:= Boolean ∪ Integer ∪ Real ∪ String.
$view_i$	denotes a view.
${}^{view_i}D$	is a set of class definitions and predicate definitions over the alphabet ${}^{I}L$. It is defined in a view.
${}^{view_i}R$	is a set of rules. It is defined in a view.
X	A set of dataflow variables of a P-KARL language.
X_E	a finite set of element variables
X_V	a finite set of value variables
X_Σ	a finite set of class variables.
ψ	formula
ψ^A	the interpretation of a P-KARL formula ψ.
ϕ	formula
Π	a finite set of predicate symbols.
Σ	a finite set of class names.
Λ	denotes an L-KARL clause.
+	built-in predicate in L-KARL

$<$ mathematical symbol or built-in predicate in L-KARL

ν a variable-assignment which can be extended to a term assignment

$\wedge$ logical and.

$\vee$ logical or.

$\neg$ logical negation.

$\leftarrow$ logical implication.

$\exists$ existence quantor.

$\forall$ all quantor.

Θ denotes a substitution

$\leq$ is-a relationship between classes or a partial ordering.

$\in$ is-element of relationship between elements and classes.

$\cong$ equality relationship between object denotations.

: connect a single-valued attribute with its value.

:: connect a set-valued attribute with its value.

$\leq_U$ is a partial ordering on U.

$\equiv$ logical equivalence of two formulae: $\phi \equiv \psi$.

$|=$ L-KARL: I $|= \phi$ if the formula ϕ is satisfied by the semantical structure I. S $|= \phi$ if every model of the set of formulae and definitions S is also a model of the formula ϕ.
P-KARL: $A,\nu |= \psi$ iff $\nu \in \psi^A$.

$|=_u$ I $|=_u \phi$ if the formula ϕ is satisfied by the untyped semantical structure I. S $|=_u \phi$ if every untyped model of the set of formulae and definitions S is also an untyped model of the formula ϕ.

$|\neq$ opposite of $|=$

$|=_\nu$ I $|=_\nu \phi$ if the formula ϕ is satisfied by the semantical structure I and the term assignment ν.

$|\approx$ S $|\approx \phi$ if the untyped perfect model of of a set of formulae and definitions S is well-typed and also a model of the formula ϕ.

$|i|$ set of facts which are given by the input-data part of a domain layer.

$|o|$ set of facts which are contained by the output-data part of a domain layer.

$|store_i|$ set of facts which are the current content of the store $store_i$.

$\|T\|$	denotes the equivalence class of “T” which is induced by the equality terms.
<=+=	a positive edge in a dependency graph $DG_L(S')$
<=–=	a negative edge in a dependency graph $DG_L(S')$
<<	a preference quasi-order “<<“ on all models of a set of rules and definitions S.
$\cup$	nondeterministic choice in a P-KARL program or the set-union operator.
;	sequence in P-KARL program.
*	iteration in a P-KARL program.
?	test-operator in a P-KARL program.
$\wp(A)$	power set of A.
$\wp_{\uparrow \leq_A}(A)$	denotes the subset of $\wp(A)$ containing all upwards-closed subsets according to $\leq_A$.
True(_)	unary predicate symbol of a P-KARL language.
$\varnothing$(_,_)	binary predicate symbol of a P-KARL language.

List of Tables