Technische Universität Berlin
FACHBEREICH INFORMATIK
INSTITUT FÜR ANGEWANDTE INFORMATIK

COMPUTERGESTÜTZTE INFORMATIONSSYSTEME
LEITUNG: PROF.DR. H.-J. SCHNEIDER

PROJEKTGRUPPE KIT
KÜNSTLICHE INTELLIGENZ
UND TEXTVERSTEHEN

SEKR. FR 5-8
FRANKLINSTR.28/29
D -1000 BERLIN 10

KIT BACK

KIT-REPORT 41

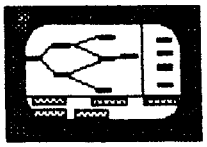
The Anatomy of the BACK System

Kai von Luck
Bernhard Nebel
Christof Peltason
Albrecht Schmiedel

January 1987

This work was partially supported by the EEC and is part of the ESPRIT project 311 which involves the following participants: Nixdorf, Olivetti, Bull, Technische Universität Berlin, Universita di Bologna, Universita di Torino and Universität Dortmund

ISSN 0931-0436



The Anatomy of the BACK – System

Kai von Luck
Bernhard Nebel
Christof Peltason
Albecht Schmiedel

TU – Berlin
Project KIT – BACK
Sekr. FR 5 – 8
Franklinstr. 28/29
1000 Berlin 10

Jan. 1987

1 Introduction

The BACK system[1] is being developed at the Technical University of Berlin, Computer Science Department, by the KIT-BACK group[2]. BACK is an integrated hybrid knowledge representation system based on the ideas of KL-ONE (s. [Brachman & Schmolze 85]) with well-defined and operationalized semantics.

The major topics in developing the BACK system were the close integration of two different knowledge representation formalisms and the careful selection of representation constructs with respect to tractable interpretation algorithms.

The close integration approach arises from the experiences taken from loosely coupled system like KL-TWO [Vilain 85] or NISRL [Emde et al 84]. Thus BACK is motivated as an *integrated hybrid system* with *balanced expressiveness* of the different knowledge representation formalisms.

Arguments will be made to motivate the specific selection of language constructs done for the BACK system especially under considerations of the *tractability* of interpreter algorithms as pointed out in e.g. [Brachman & Levesque 84] and [Patel-Schneider 86].

Additional efforts were undertaken in developing user-friendly interfaces for inspecting, modifying and maintaining models of a selected domain. This line of research was motivated by the user experiences of KL-ONE-based system. For example, a number of concrete demands arose from using BACK as a system for modeling a larger domain in a consistent manner (cf. [Schmiedel et al 86]).

---

[1] Berlin Advanced Computational Knowledge Representation System

[2] A former, in various aspects different version of BACK was described in [Luck et al 85].

In order to provide the reader with a self-contained document, this report gives an introduction to the overall philosophy of KL-ONE and hybrid systems based on KL-ONE. That means no deeper knowledge about these topics is presupposed, although this may be helpful.

A more theoretical view on the formalisms for representation of knowledge offered by the BACK system is taken in chapter 3. This includes the definition of the formal syntax and semantics of these formalisms and critical discussions of this approach including comparisons with other current work.

In chapter 4 the different aspects of the BACK system as a knowledge management system are described and the specific design decisions made in BACK are presented and discussed.

The BACK system user interface is briefly presented and motivated in chapter 5.

Additional to this report, **The User's Guide to the BACK System** [Peltason et al 87] is available for an introduction to the practical use of the system.

The BACK system is fully operational running under different Prologs on Symbolics 36xx and IBM 4381. It was tested up to the present with several small and one more realistic domain. First results are also reported in [Schmiedel et al 86]. But the BACK system is still an experimental system and by no means a final product.

2 Representation Philosophy

Knowledge representation and the development of suitable formalisms has always been a major concern in the history of Artificial Intelligence. In the last years, the focus on this subject has increased even more, since efficient utilization of large quantities of world knowledge is regarded as the key to building intelligent systems. *Knowledge-based system* has almost become a synonym for *AI system*.

The BACK system described in this report is a close relative of KL-ONE, a knowledge representation language introduced by Brachman in 1979 [Brachman 79]. KL-ONE was motivated by the problems encountered in the semantic network tradition of knowledge representation, so a short discussion of these seems to be a suitable starting point.

2.1 Semantic Nets

One of the most common paradigms of knowledge representation in AI are semantic networks. Originally introduced by psychologists ([Quillian 68], [Norman & Rumelhart 75], [Anderson & Bower 73]) as models of human memory, they quickly became popular in AI for a number of reasons:

- Representing concepts, objects, and situations as nodes and relations between these as arcs (or links) seems to be a natural way of capturing the essence of every-day knowledge. Nets are easily visualized graphically, and lend themselves to computer implementation via well-known node-and-pointer data structures.
- Class/subclass/element relationships are easily expressed in semantic nets by a special type of link, mostly called *IS-A* link; property inheritance from classes to subclasses and

instances can thus be performed simply by following these links.
- Extracting knowledge and drawing inferences on the basis of nets may be performed by a set of special-purpose procedures tuned to the different kinds of nodes and links in the net.

Various systems were able to demonstrate the utility of the semantic network paradigm ([Raphael 68], [Simmons 73], [Carbonell & Collins 74], [Woods et al 76], [Hendrix 79]). But as part of a more general discussion concerning the foundations of knowledge representation in AI which set in around the mid-seventies (cf. eg. [Woods 75], [McDermott 76], [Hayes 77]) severe shortcomings of the semantic network approach in vogue until then were identified.

These problems all more or less revolved round one central issue: What do nodes or links in a semantic network mean exactly? Or, put in other words, the semantics of semantic nets themselves were at stake. In the end, these questions boil down to what semantic networks offer to a user. Is it more than just a clever data strucure and a set of more or less useful routines for accessing them?

The problem of giving semantic nets a precise semantics is discussed for example in [Woods 75], [Brachman 77], and [Brachman 83]. Their work demonstrates that there are potentially many different interpretations of links and nodes in a network. Unfortunately, some of these were confused even in the design of a single network formalism[3]. In order to perform deductions on the knowledge encoded in a semantic network, there must be a fixed meaning of the nodes and links in the network independent of the procedures accessing the network. In case the intended meaning is not made precise, there is no other way but

[3] It should be noted that this is unfortunate only from the viewpoint of research interested in computational knowledge representation schemes that are of practical use in larger AI systems. From a cognitive modeling point of view (where semantic nets originated in the first place), things may appear quite different.

to look at the code of these procedures. We end up with semantics in terms of specific set of procedures. This may be sufficient for solving a particular problem, but certainly will not do for knowledge representation formalism with some claim of generality.

Possible meanings of links depend on what the nodes stand for. Nodes in semantic nets have been taken to represent a variety of things: concepts, sets, predicates, prototypes, natural kinds, individuals, abstractions, and propositions, just to name the most important ones. An obvious distinction is the one between nodes representing individuals and generic nodes standing for or describing many individuals or classes. In early semantic nets, even this distinction was blurred; for example, an IS-A link was used both for linking a generic to another generic as well as an individual to a generic. This amounts to the confusion of the subset and member relationship, or, if nodes are viewed as predicates, the confusion of a quantified formula (forall x: p(x) --> q(x)) and the application of a predicate to an individual (p(a)).

A very important categorization of the possible meanings of links concerns their **assertional** impact: does the presence of a link automatically imply the truth of an associated fact in the world? If generic nodes are construed as atomic predicates or sets and links between them as universally quantified formulas (or subset relationships) this seems to be the case. An IS-A link between *canaries* and *birds* simply asserts that the set of canaries is a subset of the set of birds. This seems to be a straight-forward fact about the world, but consider another example: *a four-legged canary IS-A canary.* Although this is probably a correct assertion, this example suggests another possible interpretation of the IS-A link, e.g. a non-assertional one. In this case, no information whatsoever is conveyed about the world; the link merely expresses the conceptual containment relation between *four-legged canary* and *canary* which holds utterly independent of anything true or false in the world. Or,

in other words, the IS-A link is justified purely on the basis of the structure of the descriptions involved, not on any way the world happens to be. We will elaborate on this approach later, as this is the route pursued in KL-ONE, and, following KL-ONE, in BACK.

By far the most common use of IS-A links is to realize some kind of property inheritance. Properties shared by a set of individuals are linked to a node representing the set; all nodes connected via IS-A links (and thus introduced as subsets) inherit them. When links and nodes in a network are given a clean first order logical interpretation, there is no real problem: the network is simply a notational variant. But, unfortunately, usually there is no such clean interpretation. Instead, the typical use of IS-A in inheritance networks is tied to the notion of inheritance with exceptions: properties connected to a node at one level may be cancelled at a node below. This is certainly not compatible with interpreting IS-A links as univerally quantified conditionals and at the same time interpreting property links as universally quantified as well. In fact, Brachman [Brachman 83] has conclusively demonstrated that networks admitting inheritance with exceptions cannot be meaningfully interpreted as representing generic concepts at all.

This is not to say that these kinds of inheritance schemes are useless or even fundamentally wrong. In fact, inheritance with exceptions has received much attention in research and formalisms and calculi have been designed to provide a formal semantic foundation [Touretzky 86]. Also, there is no doubt that knowledge about prototypical properties is an important part of representing conceptual structures. But, taken as the backbone of a representation system claiming to deal with knowledge on a conceptual level, there are fundamental drawbacks. Representation of prototypical properties, possibly using some kind of formalism for inheritance with exceptions, should be clearly separated from representing the concepts themselves.

In addition, even when inheritance with exceptions is not admitted, major problems remain with all formalisms which adher to the assertional interpretation of generic nodes and links. This is due to the fact that in these formalisms there is no way to treat the intensional character of conceptual knowledge. Thus, it is impossible to give a proper account of composite descriptions: using concepts to form new, more complex concepts the meaning of which is canonically derived from the structure of the complex description.

2.2 Levels of Representation

A major breakthrough concerning the semantic foundation of semantic nets was due to R. Brachman, who introduced what he called the **epistemological level** [Brachman 79]. Maybe the term is not very well chosen because there is no obvious connection to what is usually referred to as epistemology, but Brachman introduces it to characterize a distinct level of knowledge representation which had not received enough attention of researchers in the field until then. His objective was to clarify some of the confusions that had arisen from mixing various levels of representation, and to propose a new, more promising approach. Based on the different approaches taken until then, he distinguishes four levels of representation each of which suggest a set of **primitives** for semantic net languages. He stresses the need for identifying these primitives because only if they are fixed and understood in advance a fixed interpreter can be built. The primitives are the elements that the interpreter 'understands' and that are not explicitly represented in the network itself. They determine together with a set of rules that are realized by the interpreter which non-primitive elements may be formed. These levels are:

- **implementational level:** When semantic nets are defined at the level of pointers and nodes, then they are obviously merely a kind of data structure with no claims about possibly useful ways of structuring knowledge. Also, there are no inherent constraints for interpreters for these kinds of nets.
- **logical level:** Network primitives are regarded as a set of logical primitives such as AND, OR, and THERE-EXISTS. In this case, an interpreter would be required to conform to the logical meanings of these primitives. A semantic network in this perspective boils down to a particular implementation of logic enriched with some kind of indexing mechanism.
- **conceptual level:** This seems to be the classical view of semantic networks. Primitives for these nets are a small set of primitive concepts (object- and action-types) and primitive conceptual relations (deep cases) thought to be language-independent building-blocks out of which all other concepts should be constructable. Schank's conceptual dependency nets using primitives such as PTRANS, MTRANS, GRASP, INGEST, INSTRUMENT, RECIPIENT, AGENT, etc. are a typical example.
- **the linguistic level:** Network primitives are taken to be language-specific words and expressions, but this approach seems very rarely to have been adopted.

How are these levels related? Obviously, given a knowledge-based system as a whole, it can be analysed on any of these levels at the same time. The problem is not one of choosing the *correct* level; rather, it is the way the levels are related. The problem with earlier semantic nets was that the levels were not clearly kept apart, freely intermingling primitives from different levels in one formalism. If a formalism is to be useful as a tool, it should be clear at what level its primitives are located, and it should be neutral with respect to what can be built on top of them. For instance, a formalism at the logical level can (and indeed should be) independent of a particular set of conceptual primitives.

But back to Brachman's proposal concerning the missing **epistemological** level. He suggests a set of primitives constituting an intermediate layer between logical and conceptual

level. This level is concerned with using concepts as intensional descriptions. Brachman stipulates the existence of a small set of relations relating parts of an intensional description to a whole which are not accounted for as primitives at the logical level. Furthermore, they are definitely below the level of conceptual primitives insofar they are not committed to any particular set of these. Intensional descriptions can be related solely by virtue of their internal structure. In addition, a rigorous treatment of inheritance can be given on this level. Inheritance of some sort or another is common to all semantic networks, although it is neither a logical primitive, nor is it properly accounted for by a 'semantic' relation on the level of conceptual primitives. Giving concepts internal structure and relating concepts in terms of these allows for a notion of structural inheritance on the basis of conceptual containment, but rules out inheritance of default properties, which are more properly dealt with elsewhere.

Another important effect of treating concepts as intensional descriptions is the ability of forming composite descriptions in a syntactically and semantically well-defined manner, something former semantic nets were unable to do.

2.3 The Hybrid Approach to Knowledge Representation

Brachman's analysis of the different levels of representational primitives and his proposal of an distinct epistemological level concerned with intensional descriptions was the starting point of KL-ONE, which soon became a kind of paradigm sparking off substantial research efforts. Several workshops bringing together people working on this paradigm were held (e.g [Schmolze & Brachman 82], [Moore 86]). A number of experimental systems were built, each focussing on different aspects of KL-ONE, but all relying on a core of basic ideas.

Maybe the most important of these is the commitment to a radical distinction of **terminological** and **assertional** knowledge, a distinction which gave rise to the term **hybrid** knowledge representation systems. These kinds of systems provide two distinct sets of representational primitives, each of which are associated with a well-defined syntax and semantics, and a class of inferences which are at least sound and more or less complete[4] with respect to their semantics. The terminological level is concerned solely with a set of intensional descriptions formed by a small set of concept-forming operators (the actual representational primitives of this level) and their taxonomic relationships which are determined only by their structural properties. These taxonomic relationships have no assertional import on their own; but descriptions can subsequently be used to make assertions about the world, and, by virtue of their structure, additional inferences may be drawn. Complex descriptions can be associated with names, but these are pure definitions and carry no additional meaning beyond the descriptions they abbreviate. The system component responsible for maintaining descriptions and their taxonomic relationships is usually referred to as **TBox**, wheras the component dealing with the facts of the world (assertional knowledge, contingent facts) is called **ABox**.

Another common feature which is a consequence of the representational primitives chosen for the terminological level is the neutrality with respect to semanctic or conceptual primitives. The primitives provided for structuring concepts and descriptions do not in any way determine the domain- or language-specific primitives. These can freely be chosen and implemented in terms of the epistemological primitives. In other words, no ontological assumptions (beyond the distinction of terminological and assertional knowledge) are made.

---

[4] Completeness **and** tractability is usually very difficult to achieve simultaneously, at least for formalisms with reasonable expressive power. We will repeatedly recur to this problem within this report.

2.3.1 The Choice of Primitives for the Terminological Level

During the development of KL-ONE, various proposals for particular sets of epistemological primitives have been made. The early ones, including Brachman's own, relied heavily on a graphic, node-and-link type of notation revealing the heredity of the semantic network tradition. Later on, beginning with [Schmolze & Israel 83], a linear syntax was introduced, which, although maybe less illustrative, faciliates formal analysis.

For all of these, the basic entities all structured descriptions are composed from are **concepts** and **roles**. Concepts and roles are arbitrary descriptive terms, where concepts are applicable to objects, and roles to relations between objects. When using concepts or roles for making assertions about objects, concepts correspond to one-place and roles to two-place predicates. All concepts and roles fall into two distinct groups: **primitive**[5] or **defined**. The meaning of a defined concept (or role) is completely determined by its internal structure: the parts the concept is composed from specify necessary and sufficient conditions, whereas primitive concepts are only partially determined by their structure, which in this case provides only necessary conditions.

Internal structure of concepts is achieved by composing concepts with a small set of term-composing operators. In the following, we give a short overview of main operators chosen for the TBox of the BACK system together with some examples which are typical for all KL-ONE alikes. We will use a linear notation, but at the end of this chapter we give an example in graphic notation and explain the mapping between the two. Consider the following examples[6]:

[5] Note that 'primitive' here refers to kinds of concepts and roles; previously, we used 'primitive' for identifying the basic elements of a representation formalism.

[6] The complete syntax and semantics of TBox expressions in the BACK system is introduced more formally in chapter 3.1.


```
Thing          = rootconcept1
Person         = primconcept1(specializes(Thing))
Male           = primconcept2(specializes(Thing))
Female         = primconcept3(specializes(Thing))
offspring      = primrole1(domain_range(Thing,Thing))
married_to     = primrole2(domain_range(Person,Person))
child          = primrole3(domain_range(Person,Person),
                           differentiates(offspring))
Parent         = defconcept(specializes(Person),
                           nrmin_restriction(child,1))
Mother         = defconcept(specializes(Parent),
                           specializes(Female))
Mother_of_Daughters =
                 defconcept(specializes(Mother),
                           value_restriction(child,Female))
Woman          = defconcept(specializes(Person),
                           specializes(Female))
Married_Person =
                 defconcept(specializes(Person),
                           nrmin_restriction(married_to,1))
Unmarried_Person =
                 defconcept(specializes(Person),
                           nrmax_restriction(married_to,0))
Wife           = defconcept(specializes(Married_Person),
                           specializes(Female),
                           value_restriction(married_to,Male))
Bachelor       = primconcept4(specializes(Unmarried_Person),
                           specializes(Male))
Unmarried_Man_with_three_children_all_of_them_Female =
                 defconcept(specializes(Unmarried_Person),
                           specializes(Male),
                           nrmin_restriction(child,3),
                           nrmax_restriction(child,3),
                           value_restriction(child,Female))
```

These expressions assign concept (or role) definitions to names. The definitions (right-hand side expressions) are built from term-forming operators[7], and from names, which are place-holders for their respective definitions. Names have no meaning on their own; they are only for convenience of the user. For example, the definition for Parent could be expanded by replacing all names by

[7] Throughout this chapter, term-forming operators are in bold, concept names begin with an upper case and role names with a lower case letter.

their definitions, so that the expanded form contains only term-forming operators:

```
Parent = defconcept(
            specializes(
               primconcept1(
                  specializes(rootconcept1))),
            nrmin_restriction(
               primrole3(
                  domain_range(.....),
                  differentiates(
                     primrole1(
                        domain_range(.....)))),
               1))
```

The indices at **rootconcept**, **primconcept**, and **primrole** indicate that each use of these operators with a new index introduces a new primitive (cf. chapter 3.1.1). With these, the actual conceptual domain primitives can be introduced. As already mentioned before, the structure of a primitive concept or role provides only necessary, but not sufficient conditions. So, given the examples above, describing someone as being a *Bachelor* necessarily implies the truth of the description *Male* and *Unmarried_Person*, being part of the structure of the definition of *Bachelor*. On the other hand, given the description *Male* and *Unmarried_Person*, this does not necessarily imply *Bachelor*. But, as *Mother* is introduced as a defined concept, describing something as being a *Parent* and being *Female* necessarily implies it is a *Mother*, because the structure of defined concepts are sufficient for recognizing an instance of the concept.

As is probably obvious from inspecting the examples, the **specializes** operator has the effect of including the concept referred to in its argument as part of the description of the concept being defined. Thus, *Parent* is part of the description of *Mother*, and *Female* is part of the description of *Woman*. By using a **specializes** term in a definition, the concept being defined is explicitly made a subconcept of the concept being

specialized, thus inheriting all of its structure. Using the **specializes** and **primconcept**$_n$ operators only, a hierarchy of primitive concepts can be defined.

The **nrmin_restriction**, **nrmax_restriction**, and **value_restriction** operators are used to define concepts in terms of their relationships to other concepts. **nrmin_restriction** and **nrmax_restriction** restrict the cardinality of role-fillers for a given role. In the definition of *Parent*, **nrmin_restriction**(child,1) specifies that there must exist at least one object as a role-filler of the child role. As this is part of the definition of *Parent*, we can infer that for anything described as *Parent*, an object in the *child* role must exists, although we may not know who it is. On the other hand, if something is described as *Person*, and additionally there is another object asserted to be in a *child* relationship, we can infer that this is an instance of *Parent* because all the conditions of the *Parent* definition are fulfilled. Of course, this last inference would not be valid if *Parent* had been introduced as a primitive concept. **nrmax_restriction**(role,n) specifies that there are at most n objects as role-fillers for role. Combination of nrmax_restriction and nrmin_restriction can be used to specify an interval of the number of permissible role fillers; an important special case is defining a role to be functional by **nrmin_restriction**(role,1) and **nrmax_restriction**(role,1).

The **value_restriction** operator is used to restrict the kinds of objects permissible as role fillers. In our examples, *Wife* must be *married_to* a *Male*. Note that a **value_restriction** does not require the existence of a role filler for the role; only if any at all exist, they must fulfill the restriction.

Roles have an internal structure much the same way as concepts. However, for the BACK system at the present stage we have restricted ourselves to primitive roles allowing for necessary, but not for necessary and sufficient conditions for a

role term. Note that in the examples all roles are introduced with the **primrole**n operator. The **differentiates** operator is the role analogue for **specializes**. As part of a role defining term, it requires the specified role to be a subrole of the role differentiated. For example, in the definition of child above the term **differentiates**(offspring) makes *child* a subrole of *offspring* incorporating the *offspring* relation as part of the *child* relation. With this operator, a role hierarchy can be set up much the same as for concepts. The **domain_range** operator restricts the domain and range of a role according to the concepts of its arguments. In the example, the *child* role requires the related entities to be describable as *Person*.

Looking at the examples above, there are obvious relationships between the concepts: the way they are defined, a *Bachelor* is necessarily a *Male*, *Wife* and *Mother* are both necessarily *Woman*, etc. In fact, for each pair of concepts, it can be determined whether one concept is **subsumed** by the other. This is trivial if one concept was explicitly introduced as a specialization of the other (via the **specializes** operator), but a concept can subsume another even if it was not explicitly defined as a subconcept. *Mother* was defined as *Female Parent*, and *Woman* as *Female Person*. But the *Person* description is part of the *Parent* description, so the subsumption relationship holds. Subsumption is computed only by inspecting the structures of the concepts involved; it can be seen as conceptual containment. According to the subsumption relationship, which defines a partial ordering on the set of concepts, these can be placed in a directed acyclic graph where links stand for the subsumption relationship (superconcept - subconcept relationship). Effectively computing this graph for a set of concept definitions such that for every concept its subsumers can be found by following links upward, and its subsumees by links downward is referred to as **classification.** Inheritance of necessary structural components of concepts to all their subconcepts is a natural side effect.

Although sound algorithms for computing subsumption are fairly simple and computationally tractable, it is difficult to come up with tractable **and** complete algorithms. This is heavily dependent on the set of term-forming operators chosen. Until now (although this may change in future), we have not included a **defrole** operator, the analogue to the **defconcept** operator for roles. With defined roles, no tractable and complete subsumption algorithm is known, at least for the usual, intuitively plausible two-valued semantics [Brachman & Levesque 84] [Patel-Schneider 87]. This operator, however, would certainly enhance the expressivity of our TBox language. For instance, the *child* role could then be a defined role, defined as an *offspring* relation between *Person*. A *son* role could be defined as a *child* role the range of which is restricted to *Male*.

Figure 1 shows some of the concepts and roles of the examples in a graphical notation. The shaded ellipses are primitive concepts, the white ones defined. The fat arrows indicate subsumption relationships; note that they do not correspond one to one to the **specializes** terms in the linear definitions because we are assuming a classified net where concepts are placed at the most specific position possible. The role hierarchy is indicated at the right; in our case there are only two: *offspring* and *child*. The role links show the restrictions for a role at a concept: the *child* role at *Parent* is number restricted to the interval 1..infinite, and at the *Mother_of_Daughters* concept it is additionally value restricted to *Woman*.

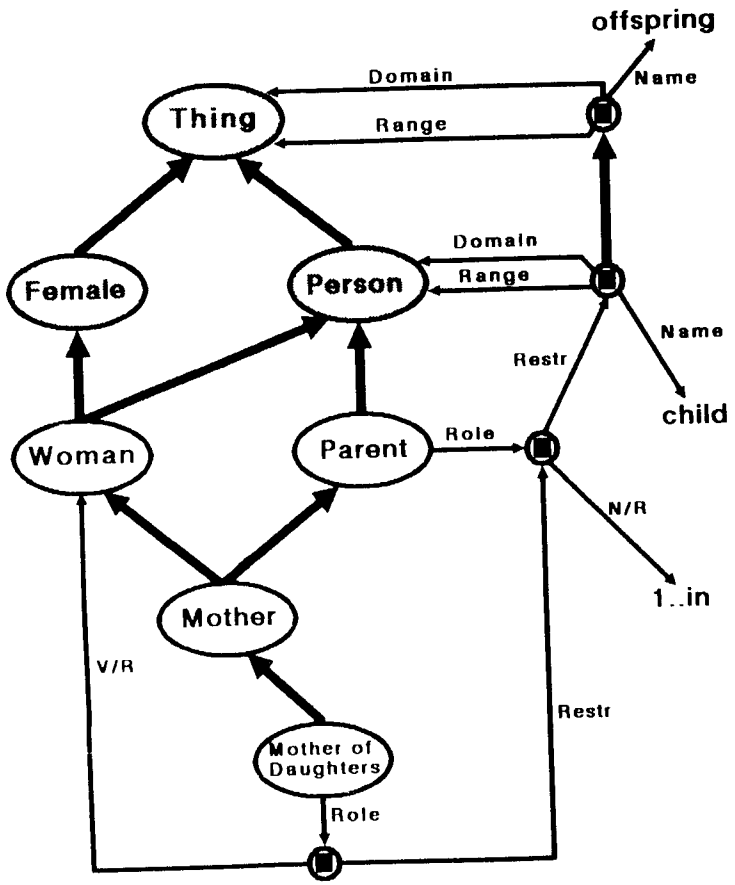


Fig. 1: Some Example Definitions in Graphical Notation

2.3.2 The Choice of Primitives for the Assertional Level

Traditionally, KL-ONE has always been more concerned with the terminological part of the hybrid representation scheme. As a consequence, there is no generally accepted set of primitives for representing assertional knowledge associated with KL-ONE. Various, widely differing proposals have been made, some of which have been implemented.

In the earliest and most simplistic approach to the assertional level (cf. [Schmolze & Brachman 82]), assertions are represented by **nexuses**, which are connected to TBox concepts by **description wires**. A nexus with a description wire denotes the existence of an object satisfying that description. Coreference of descriptions can be expressed by connecting several description wires to one nexus. Nexuses are placed into **contexts**, allowing for different 'worlds' to be represented. As far as we know, these ideas were never taken as a starting point for an implementation.

KL-TWO [Vilain 85], and KRYPTON [Brachman et al 83] were the first implementations of KL-ONE combined with a sophisticated assertional component, thus deserving to be labelled hybrid systems. KL-TWO is the combination of NIKL [Moser 83], a terminological representation system based on KL-ONE, and PENNI, a propositional inference engine based on RUP [McAllester 82]. In KRYPTON, a terminological component somewhat less expressive than NIKL was integrated with the Stickel theorem prover [Brachman et al 85], which is a full-fledged theorem prover for first order predicate logic. Both systems showed the feasibility of the hybrid approach; but for basically two reasons we adopted another direction in the design of the assertional component of the BACK system.

Foremost, there is the problem of the computational complexity associated with a complete inference mechanism for any standard logic. In our point of view, a knowledge representation formalism

intended to be useful for representing large bodies of knowledge should not suffer from the possibility of combinatorial explosion built into its basic inference mechanism. But this is the case even for propositional logic. Usually the problem is circumvented by restricting inferences, either by some arbitrary limit on resources allocated, or by using an incomplete set of inference rules. But both of these are rather unsatisfactory ways of dealing with the problem because in both cases the net behaviour of the system is at odds with the semantics of the representation language. Indeed, a language as expressive as propositional or first order logic available at a user interface for an assertional component of a knowledge representation system is highly misleading just for this reason: what the user expects from logic, he will not get from the machine, although it seems to 'understand' it (or, in case of complete inference, you will have to wait a **very** long time for an answer in any but trivial cases). Therefore, we feel that the formalism at the core of a knowledge representation system should be rather less expressive, but on the other hand computationally tractable and inferentially complete with respect to its semantics[8].

The other reason for our search for an alternative set of primitives for the ABox has to do with the interplay of terminological and assertional knowledge. Taxonomic reasoning based on TBox definitions can be very powerful, but only if there are corresponding assertional primitives available that allow to make full use of them. This is what we have called *balanced expressiveness* of TBox and ABox.

As a consequence, for representing assertional knowledge we have defined a language which is in many ways less expressive than full propositional calculus, but on the other hand includes some operators involving special kinds of quantification in order to match the possibilities of the TBox. Although we have not proven the desired properties (tractability and completeness of

[8] These issues are discussed in length in [Patel-Schneider 87], [Levesque 86].

inference) mentioned above, we feel we have found a good starting point.

Our assertional language can be briefly characterized by the following features: Assertions are made by creating **unique constants** in the ABox described with a TBox concept (which may be an arbitrary complex description, see above). This is equivalent to applying concepts as one-place predicates to constants. Furthermore, unique constants are additionally described by sets of role-value pairs, where values are constructed as **value expressions**, which are basically boolean expressions without negation containing other unique constants (the role fillers), and two additional operators, the **card(min,max)** operator, and the **cwa** operator. Disjunction in value expressions allows for a certain restricted kind of incomplete knowledge for role fillers, whereas the card and the cwa operators can be used for expressing restricted kinds of knowledge involving negation and quantification. The card operator restricts the cardinality of a set of role fillers for a role; it matches the number restriction operators nrmin_restriction and nrmax_restriction of the TBox. The cwa operator (abbreviation for *closed world assumption*) restricts the set of possible role fillers for a role to be in the set of those already mentioned, thus excluding all other known or unknown candidates. Although this does not match a TBox operator one to one, closing the set of role fillers to be considered allows for additional inferences particularly in connection with value restrictions defined for a concept in the TBox. A more detailed description of the ABox language is given in chapter 3.2 and 4.2.

2.4 Towards a Functional Paradigm for Knowledge Base Formalisms

To conclude this chapter on our representational philosophy, let us summarize the main points that guided us in the design of the BACK system.

1. In our view, the purpose of a knowledge representation formalisms such as BACK is to provide a well-defined functionality as part of a larger, knowledge based system. AI systems are designed to carry out a variety of tasks for many of which a lot of knowledge is needed. Of course, it is possible to build this knowledge right into the problem solving program itself, but this approach becomes increasingly infeasible in the face of large bodies of knowledge with rich internal structure. Seperating the actual problem solving part of an AI program from the knowledge base has distinct advantages. It provides means of structuring, storing, and retrieving bodies of knowledge in a declarative manner independent of the program using it. In addition, knowledge engineering activities such as debugging, enforcing consistency, and incremental knowledge acquisition all depend on some kind of formalism that respects the fact that knowledge is always about something in the world, and thus more than a mere data structure.[9].

2. In this perspective, it seems natural to view a knowledge base as black box with whom a user (a person or a program) communicates via a tell/ask interface. The user is not concerned with how the KB performs its task, only with its functionality which is specified by the semantics of the interface language[10]. The terms tell and ask stress the fact that communication of knowledge is always about something in the world and that the black box should behave accordingly.

3. The communication language has to be expressive enough to be able to convey useful knowledge for a broad range of domains. Expressiveness is an issue at two different levels: the

[9] Of course, from the viewpoint of a programmer, anything represented in a computer can be seen as a data structure. However, to be used as knowledge data structures must have an interpretation in the outside world, and a user must be able to rely on the assumption that these data structures are manipulated in a way consistent with their interpretation (cf. [Newell 82]).

[10] This functional view of knowledge representation was effectively put forward by [Brachman et al 83].

terminological level and the assertional level. Languages to be useful for communicating richly structured knowledge must be able to handle complex concepts in terms of their internal structure rather than treat them as primitives. In order to be able to do this, the user should be able to tell the knowledge base about the meaning of a concept. This is especially important for technical, task- and domain-specific concepts likely to occur in AI knowledge bases. Expressivity at the terminological level is determined by the primitives provided for defining new, complex concepts in terms of already known ones. On the other hand, communicating knowledge implies the use of concepts to assert facts about the world. Expressivity on this level depends on the kind of primitives provided to construct such assertions.

4. In order to successfully transfer knowledge via a language, both partners must in some sense understand the language; the language must have an interpretation that both partners share. In particular, the user of a knowledge base must have reliable information on how the knowledge base interprets what it has been told, what kind of inferences it draws, how it combines different pieces of knowledge, etc. This is accomplished by providing a well-defined, formal semantics for the tell/ask language.

5. If the semantics is to be a reliable guideline for the user, it should be fully operational: all consequences of knowledge told as specified by the semantics ought be computable, and thus subsequently be retrievable by asking the knowledge base. Formal semantics are not worth the effort if in the end they do not correctly describe the behaviour of the knowledge base.

6. A knowledge base should provide answers to its users within a reasonable time. This raises very difficult problems because even for languages of rather modest expressivity complete inference procedures tend to involve combinatorial search and thus be computationally intractable. We have the problem of determining the trade-off between between expressivitiy of a

representation language and its computational complexity. This is a major issue in current research: finding representation languages that are expressive enough to be useful and still admit sound and complete inference procedures that are tractable. In the design of the BACK system, we have taken a fairly conservative approach by restricting ourselves to representational primitives that are hopefully tractable.

3 The BACK Formalism

In this chapter the BACK formalism is introduced. The *form* (the *syntax*) which is used to represent the knowledge and the *meaning* (the *semantics*) of these forms are explained. Any system dependent aspect, however, will be neglected in this chapter. Interaction with and modification of the represented knowledge as well as the operationalization of the semantics will be described in the next chapter. This separation of issues proves to be useful in serveral respects:

- The semantics can be investigated without referring to any particular implementation. In particular, questions of *computational complexity*, and connected with that, *completeness* and *soundness* of proof procedures (relativ to the semantics) can be analyzed.
- The *expressiveness* can be determined.
- Characteristics of the formalism such as *vividness* [Levesque 86] or *balancedness* (cf. [Luck 86], [Nebel 86]) can be determined.

The description of the BACK formalism is divided into two parts according to the hybrid nature of the system (and hence the formalism). The first part gives the syntax and semantics of the language for representing the terminological knowledge (the **TBox** formalism). The second part describes the formalism for representing assertional, factual knowledge (the **ABox** formalism).

Representation of knowledge, however, is not only concerned with formalizing what is *explicitly* known, but also with what has been represented *implicitly*. Therefore, in addition to the specification of syntax and semantics, we give an informal account of *inferences* which can be drawn on the basis of the semantics. In contrast to the claim we made above, procedural metaphors are used in describing these in order to emphasize the context they are used in.

After this description the characteristics of both formalism are discussed in detail and compared with other similar formalisms. In particular the properties mentioned above are taken into account.

3.1 The Formalism for Representing Terminological Knowledge

The discussion about structural inheritance networks started off by drawing circles, little boxes and arrows between them [Brachman 79]. Because of the limited capabilities of computers in those days, the networks had to be translated into a linear form when the first implementation took place [Woods 79]. A linear form of such networks does not only offer the opportunity to feed such networks into a computer, but it makes it possible to assign compositional semantics to such networks (if the linearization is chosen appropriately).

Semantics may be regarded as superflous, in particular from a user's point of view. However, taking a scientific perspective the specification of semantics for representation languages is inevitable. Without semantics a scientific discourse cannot take place, e.g. when comparing different representation languages.

A more pragmatic consideration is that without semantics the notion of correctness is just an empty term. For example, the development of the classification procedure (cf. 3.1.3) from an intuitive-based procedure to a sound inference technique took place on the grounds of the development of semantics for KL-ONE alikes (cf. [Lipkis 82], [Schmolze & Israel 83], [Brachman & Levesque 84], [Patel-Schneider 86]), which in turn was only possible because an appropriate form of network linearization, in fact, a language, had been chosen.

Such languages are commonly termed *frame description languages* [Brachman & Levesque 84] or *term definition languages* [Schlumberger 85]. These names emphazise the fact that they are intended to be used for *defining* or *describing* categories.

However, it is not possible to make statements about *restrictions* occuring in the domain, i.e. to introduce axioms. That means a collection of expressions of a term definition language does not exclude any model (in the model theoretic sense).

At first sight this might seem to be a serious restriction for a knowledge representation formalism, even if it is only used to introduce terminology. However, with such formalisms it is possible to draw taxonomic inferences, which proved to be very powerful and which is heavily exploited in AI systems using such knowledge representation formalisms (e.g. [Nebel & Sondheimer 86]).

Additionally, BACK provides the possibility of introducing restrictions into the terminological knowledge base. This is very similar to NIKL [Robins 86], but we tried to introduce such restrictions at well defined points, excluding some very weird constructions. These restriction statements might emigrate to another component, the inferential box (**IBox**) or natural law box (**NBox**) (cf. [MacGregor 86] and Brachman's statement in [Moore 86]).

3.1.1 Syntax of the TBox Formalism

In the following the concrete syntax for TBox expressions is given in BNF[11]. The language defined does not only describe the form of the represented knowledge abstractly but is also (part of) the interface language to be used in interacting with the system. However, there are some exceptions which are marked as such.

---

[11] Nonterminals are printed with the regular font, terminal symbols are printed with **boldface**. ':==' and '|' are used as meta-symbols.

TBOX EXPRESSIONS

```
TBoxExpr        :== TBoxRestriction |
                    TBoxDefinition
```

A knowledge base consists of a set of TBox expressions, which in turn are either restrictions or definitions.

TBOX DEFINITIONS

```
TBoxDefinition  :== Name = TBoxTerm
Name            :== PrologAtom
```

A TBox definition assigns a TBox term to a name. This name can be used in the subsequent expressions to refer to that term. Because the entire system is implemented in Prolog, the syntactic category of names is just the category of Prolog atoms.

TBOX TERMS

```
TBoxTerm        :== Aset | Concept | Role
```

A TBox term is either an attribute set, a concept or a role[12]. In the semantics attribute sets and concepts are viewed as one-place predicates and roles are viewed as two-place predicates.

[12] Of course, instead of explicit TBox terms names may be used, if they are defined as appropriate TBox terms. This will become obvious from the following definitions.

ATTRIBUTE SETS

```
Aset            :== attrset ( AttributeList ) | Name
AttributeList   :== Attribute |
                    Attribute , AttributeList
Attribute       :== PrologAtom
```

An attribute set can be viewed as an extensionally defined concept. Its extension is defined by enumerating all elements which belong to the set[13]. In contrast to regular, intensional concepts, attribute sets are not related to other concepts by roles (see below). This language construct, which is unique for TBox languages, is especially useful for representing perceivable qualities, as for example colors or sex.

In other TBox languages it is necessary to introduce concepts for sexes which have very weird extensions, e.g. the extension of the concept FEMALE-SEX seems to be either the concept itself (which does not fit into the semantics), or a unique constant, which can only be decribed by being the unique member of the extension, or a set of arbitrary FEMALE-SEXes which probably does not meet the intended meaning.

The real problem with the above concepts is that they are not concepts in the usual sense, but they behave more like constants, particular individuals of the domain. However, TBox languages usually do not permit to introduce such constants, even if they are considered to be part of the terminology.

[13] That means, here we are *restricting* the domain by specifying part of the extension, i.e. we are asserting the existence of individuals, contrary to the claim made above that TDLs are not restrictive. And as a matter of fact, attributes are not really belonging to a TDL. However, they are very useful for the purpose of counting, reasoning about disjointness etc. and therefore we consider them as part of the terminological language. Perhaps, they should also emigrate to the IBox or NBox.

CONCEPTS

```
Concept       :== DefConcept ¦
                  PrimConcept
PrimConcept   :== rootconcept_i ¦
                  primconcept_j ( CSpecList ) ¦
                  Name
DefConcept    :== defconcept( CSpecList ) ¦
                  Name
```

A **concept** is either **defined** or **primitive.** The former means that the concept is totally determined by its specification (the **concept specification list**). The latter means that it is either one of the basic root concepts, which are mutually disjoint, or it is a partially specified concept mentioning only necessary conditions.

The indices $_i$ and $_j$ are not part of the system's interaction language; however, each time a root concept form or primitive concept form is entered into the system, a new root or primitive concept is generated. That means, even if no index is used, internally a mechanism generates such indices when such a form is entered. For example, after entering the following TBox expressions into the system:

```
object = rootconcept
action = rootconcept
```

from a descriptive point of view the following TBox expressions are part of the TBox:

```
object = rootconcept1
action = rootconcept2
```


CONCEPT SPECIFICATIONS

```
CSpecList     :== CSpec ¦ CSpec , CSpecList
CSpec         :== specializes( Concept ) ¦
                  value_restriction( Role , ConceptOrAset ) ¦
                  nrmin_restriction( Role , Number ) ¦
                  nrmax_restriction( Role , Number ) ¦
                  rvm( RvmOp , [ Role ] , [ Role ] )
RvmOp         :== =
ConceptOrAset :== Concept ¦ Aset
Number        :== 0 ¦ PositiveInteger ¦ in
```

A concept is defined by giving a list of concept specifications (except for root concepts, which are introduced as such). A concept specification is

- a **specializes** term, meaning that the concept is a specialization of the argument[14],
- a **value restriction** of a role (see below), with the meaning that the role is filled with individuals satisfying this restriction,
- a **number restriction** of a role giving a minimal or maximal number of role fillers, which can be 0 (zero), in (infinite) or a positive integer,
- or a **role value map** which establishes relationships between role fillers.[15]

An important point which distinguish term definition languages from other frame or network formalisms is that the concept specifications are seen as definitions. That means that concept specifications do not state that a concept *has* certain properties, but that the specified properties *are definitional* (sufficient and necessary) for this concept. Therefore, if we want to find out whether one concept is more special than

---

[14] This is similar to ISA or AKO in other semantic network formalisms.

[15] We expect to have more elaborated role value maps similar to NIKL in future versions. For this reason the two role arguments are already (one-element) lists instead of pure roles.

CONCEPTS

```
Concept       :== DefConcept ¦
                  PrimConcept
PrimConcept   :== rootconcept_i ¦
                  primconcept_j ( CSpecList ) ¦
                  Name
DefConcept    :== defconcept( CSpecList ) ¦
                  Name
```

A **concept** is either **defined** or **primitive.** The former means that the concept is totally determined by its specification (the **concept specification list**). The latter means that it is either one of the basic root concepts, which are mutually disjoint, or it is a partially specified concept mentioning only necessary conditions.

The indices $_i$ and $_j$ are not part of the system's interaction language; however, each time a root concept form or primitive concept form is entered into the system, a new root or primitive concept is generated. That means, even if no index is used, internally a mechanism generates such indices when such a form is entered. For example, after entering the following TBox expressions into the system:

```
object = rootconcept
action = rootconcept
```

from a descriptive point of view the following TBox expressions are part of the TBox:

```
object = rootconcept1
action = rootconcept2
```


CONCEPT SPECIFICATIONS

```
CSpecList     :== CSpec ¦ CSpec , CSpecList
CSpec         :== specializes( Concept ) ¦
                  value_restriction( Role , ConceptOrAset ) ¦
                  nrmin_restriction( Role , Number ) ¦
                  nrmax_restriction( Role , Number ) ¦
                  rvm( RvmOp , [ Role ] , [ Role ] )
RvmOp         :== =
ConceptOrAset :== Concept ¦ Aset
Number        :== 0 ¦ PositiveInteger ¦ in
```

A concept is defined by giving a list of concept specifications (except for root concepts, which are introduced as such). A concept specification is

- a **specializes** term, meaning that the concept is a specialization of the argument[14],
- a **value restriction** of a role (see below), with the meaning that the role is filled with individuals satisfying this restriction,
- a **number restriction** of a role giving a minimal or maximal number of role fillers, which can be 0 (zero), in (infinite) or a positive integer,
- or a **role value map** which establishes relationships between role fillers.[15]

An important point which distinguish term definition languages from other frame or network formalisms is that the concept specifications are seen as definitions. That means that concept specifications do not state that a concept *has* certain properties, but that the specified properties *are definitional* (sufficient and necessary) for this concept. Therefore, if we want to find out whether one concept is more special than

---

[14] This is similar to ISA or AKO in other semantic network formalisms.

[15] We expect to have more elaborated role value maps similar to NIKL in future versions. For this reason the two role arguments are already (one-element) lists instead of pure roles.

another, it is not sufficient to test the transitive closure of the specializes relation, but the entire specifications have to be checked. We will discuss this issue further in 3.1.3.

ROLES

```
Role          :== primrole_k( RSpecList ) | Name
RSpecList     :== RSpec | RSpec , RSpecList
RSpec         :== differentiates( Role ) |
                  domain_range( Concept , ConceptOrAset)
```

Roles are introduced by primrole terms[16]. Roles can be further specified by giving a **domain** and a **range**, which restrict the applicability of the role, and **differentiates** terms, which establish a subrole hierachy, similar to the concept hierachy.

One may be tempted to ask why there are not **defined roles** (analogous to defined concepts). The reason for this restriction is that this would add another level of computational complexity, we are not willing to pay (cf. 3.3).

At this point a short remark about the history of the BACK system may be appropriate: The first version of the BACK system described in [Luck et al 85] neither contained an explicit role hierachy, nor a means for introducing roles explicitly. Roles were rather seen as local properties of concepts and introduced implicitly. We abandoned this view (following the NIKL development) because it left unspecified the applicability range of roles, because it permitted ambiguities for locally introduced subroles, and because explicit introduction does allow for a more elegant specification of the semantics -- i.e. it is simpler from a conceptual point of view.

---

[16] Again, the index k is not part of the system's interaction language, but this index should emphasize the fact that each act of entering a primrole term into the system creates a new unique primitive role.

The description above covers the entire term description language in BACK. As it is was said above, also some limited capabilities to express restrictions are provided. One possibility is to assert that a set of concepts are **disjoint**, i.e. they don't have any common instances. The other one is to state that a primitive concept has a singleton extension by asserting its **individuality**.

RESTRICTIONS

```
TboxRestriction          :== DisjointnessRestriction |
                             IndividualRestriction
DisjointnessRestriction  :== disjoint( PrimConceptList )
PrimConceptList          :== PrimConcept |
                             PrimConcept , PrimConceptList
IndividualRestriction    :== individual( PrimConcept )
```

One point worth mentioning is that only primitive concepts are allowed to be declared disjoint. The reason for this is that otherwise the disjointness declaration would result in the assertion that the (defined) common subconcept of two disjoint defined concepts does not exist, i.e. it would exclude concepts from being existent, although they are definable without reference to the disjoint concepts, which seems to be rather weird. A similar restriction applies to the individual restriction: Only primitive concepts can be restricted in this way, because otherwise a definitional property, such as value or number restriction has to result in a singleton extension of a concept, which is difficult to justify.

3.1.2 Semantics of the TBox Formalism

Semantics for a representation formalism prove to be useful for several reasons. An important one is that it provides a communication medium for discussing and criticizing a

representation formalism, because it is possible to abstract from system dependent idiosyncrasies. This, however, implies that the semantics is to be specified in a way, which is broadly accepted, as for example logic. As a matter of fact, Newell claimed that even though logic might not be the adequate representation formalism, it is *the* appropriate tool to investigate the *knowledge level* [Newell 82].

But even if we restrict ourselves to logic, there are still a lot of possibilities to specify the semantics of a TBox:

- An *axiomatic approach* (cf. e.g. [Vilain 85]) would translate a TBox into a set of axioms of first-order predicate logic (FOPC) referring to the model and proof theory of FOPC. That means the semantics is given indirectly. The benefit of this approach is that FOPC is well understood and accepted. The disadvantage is that it does not meet the intention and intuition of TDLs in every aspect. For example, the fact that defined concepts do not restrict anything but just isolate a certain set of individuals is not captured.
- A *model theoretic approach* (e.g. [Schmolze 85], [Brachman & Levesque 84]) would aim at describing the extensions (in fact, the structure of possible extensions) of concepts and roles. This is the most direct approach, because we just specify the TBox in terms of its extension without recurring to any other formalism. Additionally, the structure of the models are open for investigation and modification, as done e.g. in [Patel-Schneider 86].
- The *denotational approach* (cf. [Schlumberger 85]) assigns meaning to TBox terms by a denotation function mapping TBox terms to other objects, preferrable mathematical objects. An appropriate candidate would be *lambda expressions*, because they capture the intuition of definitions -- of building complex functions out of other functions (cf. [Brachman et al 85])[17]. Additionally, we are able to compare concepts by

[17] In our case only truth-valued functions, i.e. predicates, are interesting.

comparing the corresponding lambda expressions directly. However, it is still an indirect way of specifying the semantics, because we have to refer to the semantics of the lambda calculus (e.g. [Stoy 77]) and FOPC.

In some sense the three ways of specifying semantics are all equivalent for our purpose. The only difference is that intuitions about the formalisms are more obviously mirrored with e.g. the denotational approach than the axiomatic approach and that some problems are seen more easily.

In the following we will use the denotational approach to give the semantics for BACK TBox terms. The fact that names can be used to denote concepts and roles will be neglected and we assume that every occurence of a name can be substituted by its associated concept or role definition, respectively.[18]

To start off, we assume three sets of predicates (defined over some domain D):

- A set of (one-place) **root predicates** RP_i with the property that two different predicates are mutually disjoint, i.e.

  forall i,j: $i \neq j$ =>
  forall x: **not** ($RP_i(x)$ **and** $RP_j(x)$)

  and that they are disjoint from the set of all (possible) attributes.
- A set of (one-place) **primitive predicates** PP_j without any restriction,
- and a set of (two-place) **role predicates** R_k.

These are the non-logical atomic building blocks we use to create all concepts and roles. These building blocks can be viewed as the recognition functions for distinguishing individuals according to non-analytical categories[19].

[18] The attentive reader will notice that by this convention we exclude definitional cycles, and that the semantics of *inheritance* is explained by substituting names by their defining expressions.

[19] However, this is just *one* view, which is also adopted in Krypton [Brachman et al 85] and Kandor [Patel-Schneider 84]. In NIKL concept primitves are seen as restrictions over a

The denotation function **F** assigns semantics to TBox terms by mapping them to functions from a domain D (for concepts and attributes) or D x D (for roles) to truth values:

F: TBoxTerms -> [D union D x D -> {false, true}]

F is defined as follows:

F[attrset(a_1,a_2,...,a_n)] =
  lambda(x).x=a_1 **or** x=a_2 **or** ... **or** x=a_n
F[rootconcept$_i$] = **lambda**(x).RP$_i$(x)
F[primconcept$_j$(CC)] =
  lambda(x).PP$_j$(x) **and** **f**[CC](x)
F[defconcept(CC)] = **lambda**(x).**f**[CC](x)
F[primrole$_k$(RC)] =
  lambda(x,y).R$_k$(x,y) **and** **f**[RC](x,y)

The function **f**, a mapping from concept and role specifications to predicates, is defined as follows:

f[CC_1,CC_2,...,CC_n] =
  lambda(x).**f**[CC_1](x) **and** **f**[CC_2](x) **and** ... **and** **f**[CC_n](x)
f[specializes(C)] = **lambda**(x).**F**[C](x)
f[value_restriction(R,C)] =
  lambda(x).**forall** y: **F**[R](x,y)=> **F**[C](y)
f[nrmin_restriction(R,n)] =
  lambda(x).**exist**$_n$ y: **F**[R](x,y)[20]
f[nrmax_restriction(R,n)] =
  lambda(x).**not**(**exist**$_{n+1}$ y: **F**[R](x,y))
f[rvm(=,[R1],[R2])] =
  lambda(x).**forall** y: **F**[R1](x,y) <=> **F**[R2](x,y)
f[RC_1,...,RC_n] =

---

domain, at least in the forthcoming revised semantics [Schmolze 86]. Both views are, however, compatible in the sense that the inference capabilities are the same, therefore we will not further elaborate this point here.

20 The form **exist**$_n$ x: P(x) is a short hand for
there exist at least n distinct x, such that P(x), i.e.:
exist x_1,...,x_n: P(x_1) **and** ... P(x_n) **and** x_i =/= x_j
(for all 1 <= i,j <= n and i =/= j)

  lambda(x,y).**f**[RC_1](x,y) **and** ... **and** **f**[RC_n](x,y)
f[differentiates(R)] = **lambda**(x,y).**F**[R](x,y)
f[domain_range(C1,C2)] =
  lambda(x,y).**F**[C1](x) **and** **F**[C2](y)

The TBox restrictions are responsible for generating axioms, which restrict the model. The restriction *disjoint*(C_1,C_2,...,C_n) creates the following axiom:

(**forall** x: **not**(**F**[C_1](x) **and** **F**[C_2](x))) **and**
(**forall** x: **not**(**F**[C_1](x) **and** **F**[C_3](x))) **and**
... ...
(**forall** x: **not**(**F**[C_1](x) **and** **F**[C_n](x))) **and**
(**forall** x: **not**(**F**[C_2](x) **and** **F**[C_3](x))) **and**
... ...
(**forall** x: **not**(**F**[C_{n-1}](x) **and** **F**[C_n](x)))

The individual restriction *individual*(*C*) results in an axiom of the form:

forall x,y: **F**[C](x) **and** **F**[C](y) => x=y

3.1.3 Inferences in the TBox

The question now arises: What can the semantics tell us? It may help us to determine *what* can be inferred from a given set of TBox expressions, i.e. what are legimate questions and *how* these inferences can be drawn.

Because of the structure of the TBox language, only certain kinds of questions make sense. For example, we are not able to prove that a concept must have a non-empty extension, because we cannot make assertions about such properties. Another non-sensical question is to ask how many subconcepts a given concept has, because there a always (potentially) infinitely many.

As it was said above, the purpose of the TBox language is to introduce terminology by *defining* predicates. And only

relationships between these predicates or inherent properties of them can be investigated. Under this topic, **inheritance**, **subsumption** and **disjointness** play a prominent role.

As mentioned earlier, inheritance is explained by the name substitution rule. The reason for this easy mechanism is that no defaults or exceptions of inheritance are permitted (cf. chapter 2). However, the substitution of names is not the whole story. Let us consider the following simple example:

```
A = rootconcept1
R = primrole1(domain_range(A,A))
B = defconcept(specializes(A),nrmax_restriction(R,5))
C = defconcept(specializes(B),nrmax_restriction(R,1),
                              nrmin_restriction(R,1))
```

In this case, we want to inherit all information from *B* down to *C*. By substituting names by their definitions we get:

```
C = defconcept(specialize(
        defconcept(specialize(rootconcept1),
                nrmax_restriction(primrole1(
                        domain_range(rootconcept1,
                                     rootconcept1)),
                                5))),
         nrmax_restriction(primrole1(
                        domain_range(rootconcept1,
                                     rootconcept1)),
                        1),
         nrmin_restriction(primrole1(
                        domain_range(rootconcept1,
                                     rootconcept1)),
                        1))
```

Apart from the fact that the above form is a monster and that it demonstrates that naming terms is a real benefit, there is still something hidden, namely that the inheritance of properties from *B* does restrict *C* only in the way that *C* is also a subconcept of *rootconcept1*. The number restriction of the role *R* at *B* however does not have any effect, because the number restriction at *C* is stronger anyway. A reduced form, which is equivalent to the above according to the semantics, can be given as following:


```
C = defconcept(specializes(rootconcept1),
            nrmax_restriction(primrole1(
                        domain_range(rootconcept1,
                                     rootconcept1)),
                    1),
            nrmin_restriction(primrole1(
                        domain_range(rootconcept1,
                                     rootconcept1)),
                    1))
```

The example shows that if we want to know the actual restrictions of a concept we have to combine the inherited and stated properties in some way. As a matter of fact, this kind of inference is always done in the system when defining a new concept and it is referred to as **completion** (cf. [Abrett & Burstein 86]).

Now, that we know how inheritance works, there might be the question whether the *specializes* relationship exists only between concepts which are connected explicitly. For instance, let us examine the following case:

```
A = defconcept(specializes(C),nrmax_restriction(R,1))
B = defconcept(specializes(C),nrmax_restriction(R,5))
```

Obviously, *A* and *B* are both specializations of *C*. However, beyond that there is more. Intuitively, every individual, which can be categorized by the definition of *A*, i.e. which is a *C* and has at most one role filler for role *R*, can also be categorized to be a *B*, because if something has at most one role filler it also has at most five role fillers. That means that *B* is more general than *A*, although this is not explicitly mentioned by a *specializes* specification.

This intuitive notion of *more general* is mirrored in the semantics by the fact that **F**[B](x) is always true when **F**[A](x) is true, but not vice versa. That means we have an exact meaning of the *more general* relationship. Even better, we are able to formalize the decision process: If we want to know, whether *B* is more general then *A*, we only have to check:

forall x: **F**[A](x) => **F**[B](x)

The maintenance of this relationship[21], usually called **subsumption**, is one of the conceptually most important aspects in the TBox component. A so-called **classifier** takes any incoming concept and places it into the hierarchy at the right place according to the subsumption relationship to other concepts.[22] Of course, this process is also applied to attribute sets, where subsumption is decided easily by inspecting the extensions.

By this classification process a consistent hierarchy is enforced, which is an invaluable tool during the construction of a domain model. Furthermore, the classifier is used while entering new individuals into the ABox. The most specfic description will be used to categorize them and this in turn can be used during retrieval. Indeed, the latter usage can be seen as the *hierarchical pattern matching* as was proposed in KRL [Bobrow & Winograd 77], but never realized in a satisfactory way, because there was no precise definition of what KRL expressions actually meant.

Unfortunately, decision procedures for the subsumption problem are computationally tractable only for very simple term definition languages (cf. [Brachman & Levesque 84]). Therefore more expressive languages, such as e.g. NIKL, a tractable and sound, but incomplete decision procedure is used[23]. And because the BACK TBox language is designed to cover more than trivial cases, this applies here too[24].

---

21 As a matter of fact, this relationship constitutes a meet-semi-lattice. Note also, that subsumption in both direction means that the concepts are equivalent.

22 The notion of classification was first introduced by Tom Lipkis in [Lipkis 82]. A first formal treatment can be found in [Schmolze & Israel 83]. Subsequently, a lot of research effort was devoted in investigating the formal properties of this process.

23 In fact, this is true for any language which is more than plainly trivial.

24 Some notes about where inferences are missing will be given in 3.3.

It might be the case that the reader is confused by the two notions *subsumption* and *inheritance*. In particular there might the question whether they are interdependent, i.e. is it possible that after detecting a subsumption relationship between two concepts the inheritance process has to be triggered (and vice versa)? The answer is obviously negative. Subsumption is decided on grounds of all (including inherited) properties of a concept. Therefore if we discover a subsumption relationship we know that all properties of the more general concept are already present in the more special concept. That means that classification cannot infer that a concept has *additional* properties, but can only change the relationship of *immediately specializing*, or to put it more picturally, to *move* concepts around in the concept hierachy.

The last kind of interesting properties of concepts we mentioned in the beginning is disjointness, i.e. whether two concepts denote necessarily mutually exclusive sets. An equivalent question is whether the common subconcept of two concepts is incoherent, i.e. denoting the empty set or the predicate which is always false.[25]

Disjointness may arise for different reasons. Obviously, root concepts are disjoint and therefore all concepts subsumed by different root concepts are disjoint too. The same applies for primitive concepts which are marked as disjoint. Apart from disjointness introduced by roots or restrictions, it can also happen that concepts are disjoint because of definitions. For instance, the value restrictions of two concepts may be disjoint concepts, or the number restrictions may be non-overlapping intervalls. Furthermore, attribute sets are disjoint if their intersection is empty.

---

25 The BACK system does not permit to create such concepts in order to always enforce a consistent state of knowledge -- in contrast to e.g. NIKL (cf. 4.1).

3.2 The Formalism for Representing Assertional Knowledge

The ABox of the BACK-System is the part of the system where the **state of affairs** of a given domain is stored. It can be also seen as the management system for concept instances of TBox concepts.

It was designed to represent **incomplete** knowledge[26] such as the following:

Tom or Dick is the father of Mary

without telling who exactly is the father,

At least one person is a friend of Mary's

without telling who the friends are,

Tom is one of the friends of Mary's

without naming all the friends of Mary's.

The formalism for representing this kind of incompleteness is designed to allow the representation of incomplete knowledge only locally in contrast to e.g. the proposals made by H.J. Levesque [Levesque 82], permitting assertions such as:

John is married to Mary or Tom is a teacher.

This restriction is for computational reasons (cf. 3.3.1).

On the other hand, inconsistent assertions like

Susan is the father of Mary

violating a value restriction of the TBox,

Tom and Dick are the fathers of Mary

violating a number restriction of the TBox or

Tom is a teacher as well as a car

violating disjoint concept definitions of car and human in the TBox, have to be rejected by the ABox not because of syntactic ill-formedness but with respect to the definitions given in the TBox. This section gives a first overview of the ABox language and its semantics. In chapter 4, the facilities for storing and

[26] The notions of *incomplete knowledge* with respect to a (hypothetically completely described) 'world' as defined in [Levesque 82] and of *incompleteness of inference procedures* with respect to given semantics should not be confused!

retrieving assertions will be explained and discussed by specifying the ABox interface language.

Each concept instance consists of an unique identifier, called **unique constant** or **UC**, a reference to the concept it instantiates, and a set of role-value pairs. The role-value pairs are references to appropriate definitions of roles in the TBox and a structure called **value expression**, which expresses the actual values of the specific instance of a role for a specific instance of a concept.

So, if you have defined in the TBox a concept *father* as

a father is a male human with at least one child, all of which are human

you can have an instance of the concept of father as an entry in the ABox representing a specific object described being a father with some objects as his children.

The ABox of the BACK System is designed according to the principles of balancedness between TBox and ABox and vividness mentioned before and discussed later. The main attention is given to the representation of incomplete but (at least) locally consistent propositions.

3.2.1 Syntax of the ABox Formalism

A syntax describing the well-formed contents of the BACK ABox is given by the following BNF:

```
ABoxExpr         :== UCId ~ UCDescr
UCId             :== uc1
UCDescr          :== Concept |
                     Concept( RoleList )
RoleList         :== RoleValuePair |
                     RoleValuePair , RoleList
RoleValuePair    :== Role = ValueExpression
ValueExpression  :== ValueTerm |
                     ValueTerm or ValueExpression
ValueTerm        :== Value |
                     Value and ValueTerm
```

```
Value          :== card( Number , Number ) |
                   Attribute |
                   UCId
```

Concepts, roles and attributes in the syntax refer to TBox terms as decribed in 3.1.1. Unique constants (UCs) are system defined unique names. The construct *card(i,j)* stands for the representation *of* at least i, at most j *instances* being in the role-relationship to another specific entity, without naming them. The construct *card* allows the representation of e.g.

John has not more than 5 friends

without naming any one of them.

It may be surprising that the *cwa* operator mentioned in the introduction is missing from the ABox representation formalism. The reason for this is simple. Every *cwa* expression can be converted to an expression containing only *card*s, i.e. it does not contribute to the semantics. On the other hand the conversion requires exponential space so that it should not actually be performed (in the system), in particular because inferences over expressions containing *cwas* can be performed with square time complexity.

3.2.2 Semantics of the ABox Formalism

The semantics for entries in the ABox can be given by a transcription of the ABox contents in formulas of first order predicate logic with the following procedure, assuming that concepts denote 1-place predicates and roles denote 2-place predicates[27]. The general form of an ABox expression

$uc - concept(role_1 = value_1, \ldots, role_n = value_n)$

can be transformed into

$concept(uc)$ **and**
$role_1(uc, value_1)$ **and** ... **and** $role_n(uc, value_n)$

[27] That means we will use *concept* and *role* as a shorthand for **F[concept]** and **F[role]**, respectively.

The form $role_i(uc, value_i)$ can be transformed as follows:

$role_i(uc, value_o$ **or** $value_p)$ is transformed into
$role_i(uc, value_o)$ **or** $role_i(uc, value_p)$,

$role_i(uc, value_o$ **and** $value_p)$ is transformed into
$role_i(uc, value_o)$ **and** $role_i(uc, value_p)$,

$role_i(uc,$ **card(o,p)**$)$ is transformed into
exist$_o$ x: $role_i(uc,x)$ **and not(exist**$_{p+1}$ x: $role_i(uc,x))$

3.2.3 Inferences in the ABox

As in the case of the TBox, we are able to infer more than it is represented explicitly in the ABox. The properties which are interesting in the context of assertional knowledge are,

- how a given object can be described most accuratly in terms of TBox concept definitions,
- what the degree of incompleteness for a given assertion is, and
- whether assertions are consistent with the rest of the ABox and with respect to the TBox.
- whether one ABox expression is more general than another, an issue dealt with in query evaluation (cf. 4.2.2).

The first aspect is solved by a kind of taxonomic inference called **realization**, a term coined by Bill Mark [Mark 82]. The main idea is that if more information than just the concept of an individual is given, e.g. the categories of the unique constants in value expressions or other descriptions of the same individual, the individual might be described more precisely by a subconcept of the concept initially specified. To put it more formally, given an individual *i* which is initially described by the following ABox expression:

$i - C_1(R_1 = v_1, \ldots, R_n = v_n)$

then there might be a concept C_2, such that C_1 subsumes C_2 and $\mathbf{F}[C_2](i)$ = true, because the individual *i*

- is described elsewhere with another concept C_x,
- is a rolefiller somewhere else, which means it is implicitly described by the value restriction of the role it is a filler of,
- or because the actual role fillers v_1 ... v_n allow for the derivation of stronger value and/or number restrictions.

Because the concept space is a semi-lattice there is even a smallest concept of this kind. This concept might not be explicitly defined, i.e. is not a *named* term in the TBox, but getting to know this concept we know all its immediate sub- and superconcepts. The principle of deriving this concept -- the **most specialized generalization** (hereafter **MSG**) -- can be stated with few words. The only thing to do is to derive a concept specification from the ABox expression (taking into account the rules mentioned above) and to classify it (cf. [Vilain 85]). However, this process, even though it sounds simple, implies a lot of complex inferences and interdependencies which we will not explore here.

One subproblem of the inference process sketched above is the determination of the degree of incompleteness for a given value expression. Only if a value expression is **closed**, i.e. all unique constants which can potentially participate as role fillers are known, the determination of a least general specialized concepts makes sense, because otherwise it cannot be more specialized than the value restriction of the role at the initially specified concept.

The determination of the degree of incompleteness does, however, not only play a role in realization, but is also of interest for the user in general. Here, the following degrees can be distinguished:

- **inconsistent**: The expression cannot be satisfied, e.g. (a and b and card(1,1));

- **definite**: The expression determines a set of role fillers uniquely, e.g. (a and b and card(2,2));
- **closed**: The expression does not determine a unique set, but it mentions all potential members, e.g. ((a or b) and card(0,1));
- **open**: everything else, e.g. (a and b).

The class a given value expression falls into is easily determined -- with polynomial time complexity, or even linear time assuming reduced expressions.

Inconsistency is the last property we will look at. It comes in different flavors. One possibility was already mentioned above, namely that value expressions can be inconsistent. This should be distinguished from cases where the empty role filler set is denoted. An inconsistent value expression is e.g. (a and card(0,0)), which is just a contradiction according to the semantics. An expression of the form (card(0,0)), however, only specifies that there is no role filler.

Inconsistencies can also arise because a concept definition is incompatible with the actual description of an individual, because one of the basic assumption about the domain is violated (e.g. the disjointness of roots) or because one of the additional axioms is not met. Generally, inconsistencies arise if the realization creates an incoherent concept. For example, if the role *married_to* of the concept *husband*, which we assume to be restricted to the concept *woman*, is filled by an individual which can be categorized to be a *man*, then we know that the role filler is an individual which is best described by

```
defconcept(specializes(man),specializes(woman))
```

which does not create a problem in the first place, because the TBox is indifferent about the existence of hermaphrodites. However, if there is, for example, an additional axiom that states that *man* and *woman* are disjoint concepts, or if they are defined to be disjoint by attributes, then the concept above is incoherent.

We will close this section with the note that this (admittedly informal) characterization of inconsistency is more general than in the former report about the BACK system [Luck et al 85] and that it is more elegant because it is just the reflection of the semantics. In addition, it solves in a very natural way one aspect of the problem of detecting so-called 'non-local' inconsistencies as described in [Luck et al 85] -- in the formalism as well as in the actual system[28]. In order to illustrate this, let us analyze the following example:

TBox contents

```
object       = rootconcept1
human        = primconcept1(specializes(object))
man          = primconcept2(specializes(human))
woman        = primconcept3(specializes(human))
set          = rootconcept2
member       = primrole1(domain_range(set,object))
team         = defconcept(specializes(set),
                          value_restriction(member,human))
small_team   = defconcept(specializes(team),
                          nrmax_restriction(member,4))
leader       = primrole2(differentiates(member),
                            domain_range(team,human))
modern_small_team
             = defconcept(specializes(small_team),
                          value_restriction(leader,woman),
                          nrmin_restriction(leader,1),
                          nrmax_restriction(leader,1))
disjoint(man,woman)
```

[28] The problem description in [Luck et al 85] also contained an system aspect, namely the order in which assertions are entered, resulting in incompleteness of inference depending on order, a very akward situation! We ignore this issue here, albeit note: This aspect of the problem has also been solved by employing a kind of constraint propagation technique.

ABox contents

```
uc1 - man
uc2 - man
uc3 - man
uc4 - man
uc5 - modern_small_team(member=uc1 and uc2
                                   and uc3
                                   and uc4)
```

Even though there is no obvious contradiction, we can infer that there is an implicit consistency violation for the following reasons: The role filler of the subrole *leader* must be one of uc_1, uc_2, uc_3 or uc_4, which are all described by the *man* concept. However, the role should be taken by an individual of the *woman* concept, which is disjoint from *man*. That means, even though we did not obviously violate a restriction, e.g. by specifying a man for the leader role, there is no valid completion of the ABox. This implicit contradiction would be detected in the following way:

- first, *realization* would try to find a MSG for uc_5 and describe it as being a *modern_small_team* and a thing which has as its value restriction for the *member* role the *man* concept;
- second, by completion, the value restriction of the *member* role is 'percolated' down to all its subroles;
- third, a new value restriction for the subrole *leader* is constructed, which is a specialization of the original one -- *woman* -- and the new one *man*, resulting in an incoherent concept.

It should be noted that this result just popped up after drawing some straight-forward taxonomic inferences and that no expensive 'puzzle mode' inferences were involved, although the situation was described in this way.

3.3 Characteristics of the Formalisms

As it might have become obvious from the text above, there exist a lot of other systems which are similar to BACK in several respects.[29] Therefore the questions arise, *what* are the

[29] KL-ONE, KL-TWO (including NIKL and PENNI) and Krypton were

differences and *why* do they exist? Some of the design rationales for our system were already mentioned in chapter 2. Therefore we will concentrate here on matters concerning *complexity* and *balancedness*, which were not discussed in chapter 2.

3.3.1 Complexity Issues

Compuational complexity is an important issue when designing knowledge representation formalisms which are intended to be used not only as a kind of communication medium between researchers, but as a means for representing and applying knowledge inside of a computer.[30] Although it has not been dealt with for a long time in Knowledge Representation, now the importance seems to be widely acknowledged. The *Computers and Thought Lecture* of IJCAI-85 by Levesque, published as [Levesque 86], elaborated on this point and gave some hints where intractability in knowledge representation formalisms can arise and what can be done to circumvent it.

One point he focussed on was a form of representation he called *vivid*. Formally, this is a form of representation where the knowledge base is a model (in the model theoretic sense) of itself; informally, a kind of representation which comes close to representation by pictures (where e.g. disjunction or negation is hard to express). As a matter of fact, this kind of representation we find in relational database systems. It is a form where a fast answer is always guaranteed. Unfortunately, it is also a very uninteresting form of representation, because the expressiveness is very limited. However, it can serve as a reference point, i.e. one can try to achieve almost vivid

already mentioned. In addition, there are e.g. Rabbit [Tou et al 82], KNET [Freeman et al 83], Kandor [Patel-Schneider 84], Meson [Edelmann & Owsnicki 86], QUIRK [Bergmann & Gerlach 86].

30 For example, the experience with Krypton seems to prove that it is not an usable system because the general theorem prover which is used as the ABox is far too slow (cf. Brachman's report about Krypton in [Moore 86]).

representations or to reduce the representation to almost vivid forms by using logically *unsound* or *incomplete* reasoning.

The ABox language introduced in 3.2 can be regarded as almost vivid. An ABox is certainly not a model of itself, however the pictoral metaphor applies here very well. The process of filling an ABox can be interpreted as recognizing a picture in a step-by-step manner by delivering only *positive* information. Disjunctions (in role filler sets) are restricted to cases where we do not know exactly the members, a situation which can certainly occur when looking at a picture. There is no way to express disjunctions concerning categorization, but only to choose a more general description (an example for reducing non-vivid representations to vivid ones also mentioned by Levesque). In particular, one cannot express arbitrary disjunctions, which Levesque regarded as one instance of extremely non-vivid representations. Indeed, these are rarely used when describing a picture, they rather resemble the kind of logical puzzles published in newspapers. Negation is severely restricted, only the *cwa* and *card* operators allow to express that something does not hold. And even this is more a kind of positive information.

In making plausible that the ABox representation formalism comes close to a vivid representation we, of course, do not claim that this proves that it is tractable. However, it gave us a good starting point for the analysis of the inferences and we made at least plausible that the desired property, tractability, holds.

It might be worth noting, that in the first version of the ABox formalism a negation operator was present, which has been dropped later on. One reason was that it was responsible for strange results in the context of the *cwa* operator. Another, more important one was that conversion to disjunctive normal form of arbitrary boolean expressions is NP complete, without negation, however, polynomial.

Another source of combinatorial explosion could be the handling of *cwa* expressions. Converting them to forms comprising only *and*, *or* and *card* requires exponential space, and also the (set theoretic) semantics in 4.2.2 gives rise to the suspicion that exponential time or space is required because the power set comes into play. Careful analysis, however, reveals the fact that for the evaluation of such expressions (e.g. reduction and detection of subsumption as defined in 4.2.2) it is sufficient to consider the set of possible members as a kind of 'evaluation context' and reducing thereby the complexity to polynomial time[31].

Concerning the TBox, there are already a lot of papers which dealt with complexity. Starting with [Schmolze & Israel 83], which describes an abstract algorithm for subsumption and notes that the algorithm is sound but incomplete, in [Brachman & Levesque 84] it is proven that complete subsumption is intractable for languages as powerful as e.g. NIKL. Finally, in [Patel-Schneider 86] the semantics is weakened to permit complete subsumption, albeit the weak semantics does not allow for an intuitive understanding of what is subsumed.

This means that currently there is no fully satisfactory solution. Either

- the expressiveness is reduced to triviality,
- the semantics is weakened without having a good intuitive model, or
- the subsumption is incomplete.

A way out of this dilemma might be to live with incomplete inferences but try to solve easy special cases and mark others as incomplete, a solution favored by the developers of NIKL [MacGregor 86].

[31] However, this strategy is not implemented in the current system and therefore disjunctions in *cwa* expressions are not permitted.

Probably, the question comes up: Where does incompleteness arise? The culprit is the introduction of subroles. If we would leave them out, a complete subsumption procedure would simply check every subterm against all other subterms. The introduction of subroles, however, complicates the situation considerably. Combinations of subterms have to be checked, which is not done in the implemented subsumption procedure for good reasons: It would result in combinatorial explosion. For example, the following subsumption relationship between *X* and *Y* is not detected:

```
A = primconcept1(...)
B = primconcept2(specializes(A))
C = primconcept3(specializes(A))
disjoint(B,C)
Z = primconcept4(...)
R = primrole1(domain_range(Z,A))
R1= primrole2(differentiates(R),domain_range(Z,B))
R2= primrole3(differentiates(R),domain_range(Z,C))

X = defconcept(specializes(Z),nrmin_restriction(R,2))
Y = defconcept(specializes(Z),nrmin_restriction(R1,1),
                              nrmin_restriction(R2,1))
```

The concept *Y* can be characterized as follows: Because the ranges of *R1* and *R2* are disjoint, the respective role fillers are necessarily different individuals. Furthermore, because *R1* and *R2* are subroles of *R*, we known that *R* must have at least two role fillers, i.e. the concept *Y* is a specialization of *X*.

A way out of this problem might be to adopt the strategy described above. Special cases, two disjoint roles, are easily detected, and the same is true for detection of possible incompleteness. The only task is to build the specialization concept for the ranges of all subroles and testing whether this concept is incoherent.

To summarize, for the sake of efficiency we sacrificed completeness. However, if the language is already on the *other side of the computational cliff*, why are primitve subroles permitted and defined subroles are not? The reasons are manifold,

but of pragmatic nature[32]. One is that defined roles introduce even more cases where incompleteness arises, e.g. a kind of *modus ponens* reasoning (cf. [Patel-Schneider 86]), another one is that we do not know how to handle defined roles on the assertional level easily, and at last that this would introduce the problem of classifying roles.

3.3.2 Balancedness in Hybrid Knowledge Representation Systems

Hybrid knowledge representation systems employ different representation formalisms in order to represent different kinds of knowledge, which are, however, somehow connected. Whether a system is really *integrated hybrid*, i.e. *one* thing made of different ingredients, and not just a diverse collection of formalisms can be decided by investigating the *glue* which holds together the different components. This should at least consist of a

- *representational theory* (explaining what knowledge is to be represented by what formalism) and
- *common semantics* for the overall system (explaining in a more abstract manner the meaning of expressions in the different formalisms).

A necessary precondition for glueing things together is that their shapes fit together, a fact we usually take for granted. And, indeed, when designing the components in one cast -- as in our case -- they usually do. However, systems as e.g. KL-TWO were built by using two components developed independently -- NIKL as the TBox and PENNI[33] as the ABox. They are in some sense *unbalanced* as we will see below. The term *balancedness*, which is a little bit vague, could be defined by the following *principle of balancedness in hybrid representation systems*:

[32] That means, they might vanish and open the opportunity to extend the language.
[33] PENNI is an adaption of RUP [McAllester 82] to KL-TWO.

> If a representation construct in a subcomponent of a hybrid knowledge representation system suggests that its usage has some impact on knowledge represented in another component (according to the common semantics), then there should be such an impact.

Even though this sounds simple, self-evident and hardly to miss, because of the common semantics, it can be easily violated.

A hypothetical hybrid system as for instance the one sketched by Figure 2 does violate this principle. The reason is that the expressiveness of both subsystems do not match. In the example system we can represent in one formalism the location of objects with a situation index, which suggests that the situation index has a certain semantic impact. In the other component, however, a situation index is not permitted, i.e. inference rules are only applied inside of one situation. The net result is that the situation index in the former component can only be regarded as a kind of comment, which has no semantic impact, however it can be used by a program using such a system.

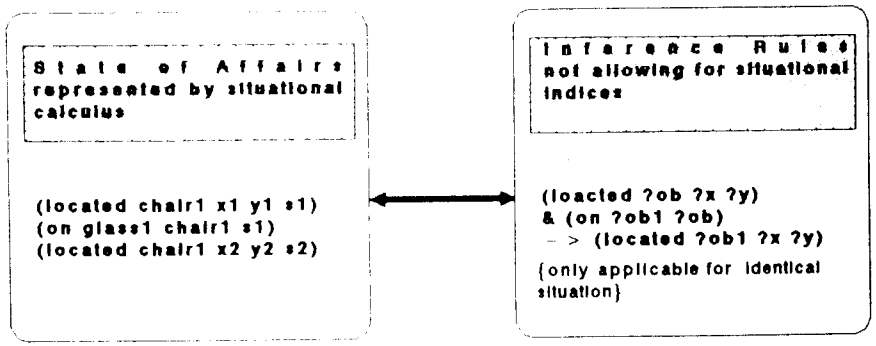


Figure 2: A hypothetical unbalanced hyprid system

With KL-TWO we encounter a similar mismatch. Because its subcomponents were developed independently, they provide operators which are used in the respective subcomponents, but without any global semantic impact. Examples are

- the *number restriction* of NIKL, which has no impact on PENNI, because in PENNI cardinalities cannot be dealt with. The reason for this is that the *unique name hypothesis* is not used, i.e. two different constants are not considered to be necessarily different. This means that role fillers cannot be counted.
- the *value restrictions* of NIKL are only used in PENNI to infer the category of a given role filler. However, the other way around does not work. A set of role fillers can never be used to infer something about the individual the role fillers are related to. The reason is that PENNI lacks an operator which allows to state that a given set is *complete*.
- the *negation* operator of PENNI is never used for the realization process, even if we get something like *not*(*Concept*(*X*)). The main reason is probably that NIKL does not provide a corresponding concept forming operator.
- the *disjunctive* operator of PENNI is not used either. If we have something like (*Concept*$_1$(*X*) *or* *Concept*$_2$(*X*)) it is certainly the case that *X* can be categorized by a concept, which is both a superconcept of *Concept*$_1$ and *Concept*$_2$. However, this is not inferred, again, probably because there is no corresponding NIKL operator.

All the above might be summarized under the topic *incomplete reasoning*. But a closer look reveals that this incompleteness has a principal reason, namely that the other component does not provide the necessary operations to realize the requested semantic structure. Usually, if incompleteness of reasoning is encountered, the solution is to write a more complete *inference procedure*, at least theoretically. The situation described above, however, cannot be solved in this way without changing the

respective *formalisms*[34]!

In contrast, the BACK system was designed to be balanced in its expressiveness. We even tried to minimize the occurence of (cross-component) incompleteness in the reasoning process. The latter design goal is the reason for only permitting primitive roles and restricting the role value maps to very simple cases. As remarked above, this is a pragmatic decision which perhaps will change in future.

---

[34] We would like to thank Marc Vilain and the NIKL group at USC/ISI for discussing the issues described here. Without it, we would not have been able to formulate our criticism.

4 The BACK System

In this chapter the main components of the BACK sytem are described. In general, we distinguish between two aspects of the work to be accomplished for the task of knowledge representation: The more theoretical point of view is concerned with specifying a representation formalism. This view emphasizes the aspect of having a sound formalization at hand which has a well-founded semantics, and for which issues like complexity, completeness of classification, etc. can be studied. This aspect was discussed under BACK *formalism* in chapter 3.

The other aspect of the representation task is concerned with the practical use of establishing and dynamically changing the corresponding part of a knowledge base. Questions in this context encompass problems of how to deal with conflicting definitions, how to delete or overwrite, how to make use of classification for the process of taxonomic reasoning, etc. This view deals with the characteristics of BACK as a knowledge representation *system*.

Many of the design considerations for development and implementation of BACK where guided by aspects of having a system for practical use; substantial emphasis was put on working out a maintainable set of functions the system should perform. Nevertheless it was of paramount importance for us to build a system which - in fact - is based on a proper formal framework as described in the previous section.

Also to be mentioned under the system view is a discussion of appropriate tools and supporting environment for all components of the knowledege representation system. All this will be discussed in the following two sections.

As shown in the diagram below, the system can be discussed following the division into

- TBox Management
- ABox Management

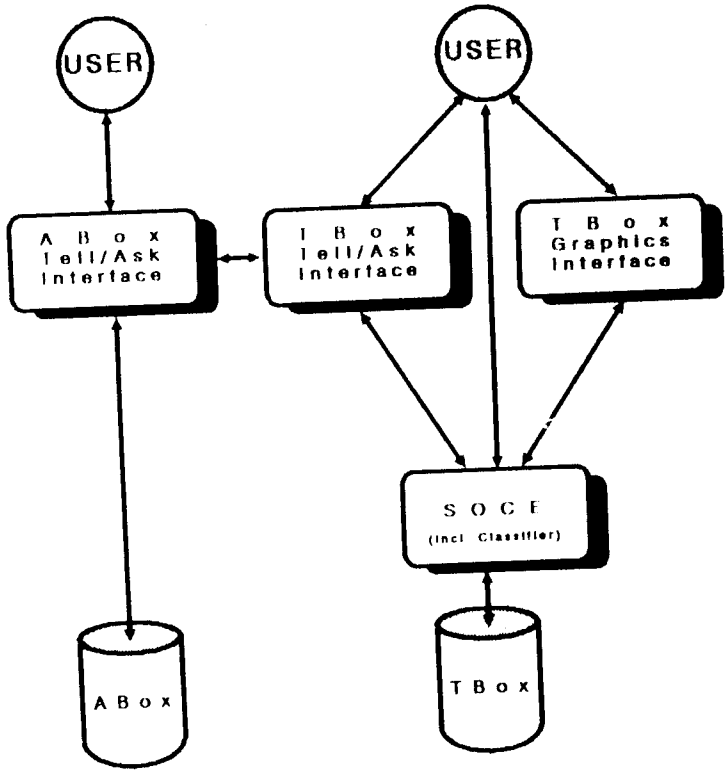


Fig. 3: BACK System Interface Diagram

4.1 TBox Management

In this chapter the terminological component of the knowledge representation system BACK, the *TBox*, is described. First, we start with a short review of the main TBox operations. The TBox contents is seen as a network structure, and the impact of operations is studied reflecting their use in the network. The set of operations discussed constitutes the functional interface for the TBox.

Then we give an overview of the system environment; the components of the system are sketched. The integration of the classifier component and its role within the reasoning process is described.

Finally, a summarizing consideration of the various net operations discussed so far leads to an extended language, the BACK TBox interface language. The specification of the language is given for interfacing with other sytem components of the overall system.

4.1.1 TBox Net Operations

In chapter 3, the BACK TBox formalism has been described as a member of the KL-ONE family of representation formalisms. In the following description we will discuss some questions which arise when the formalism is taken as the basis for a practically usable knowledge representation system.

Let us reconsider how the basic structure of the system follows directly from the structure of the formalism language: We have a hierarchy of conceptual entities, or concepts, and we have a hierarchy of 2-place relations between concepts which are called roles. The role hierarchy is a primitive subsumption hierarchy in which roles are specified by their place in the

hierarchy and by an additional specification of their domain and range. For both hierarchies the nodes are connected by links according to subsumption. Concepts and roles form two independent hierarchies; summarizing, we can see them as an overall network structure.

Also part of the network is the subnet of attribute sets: As an additional way of defining concepts we included the possibility of extensional definition by enumerating all instances of the concept explicitly in contrast to the otherwise intensional definition of concepts. All attribute sets are descendents of the predefined root concept *attributes*.

Now what problems are we faced with when we start investigating the consequences of the formalism for the dynamical management system of a such a network? First of all, we can not expect the user to come up with complete definitions (which of course, if he or she did, would have a well-defined interpretation on the basis of the formalism). Instead, the user we have in mind might prefer to start with preliminary definitions, incrementally add features to some objects, redefine, or overdefine them, finally ending up with a network he or she (or the ABox) can live with. This process has also been referred to as *knowledge editing* [Abrett & Burstein 86].

For the design of an appropriate system this means we have to make a choice regarding the way in which the incremental application of net operations is handled. We have to consider questions like: what has to be taken into account concerning inheritance, consistency, deletion, etc., when are net entities generated, and so on.

On-the other hand, we should not be tempted to open up access to all kinds of manipulation operations on the underlying data structures. In [Patel-Schneider 84] it was argued convincingly that only a minimal set of operations should be admitted as access operations in order to guarantee that the system's

behaviour meets what is specified by the semantics of the formalism.

Summarizing, we need an implementation of a well-founded representation formalism which allows for access to its objects in similar a way as in object-oriented systems concerning incremental modification and redefinition of objects. We will see how this is achieved in making some exemplary considerations on this task.

The discussion is subdivided according to the following groups of operations:

- structural net operations
- infimum / supremum operations
- marking operations
  cycles
- delete operations

4.1.1.1 Structural net operations

The first group of operations corresponds to definitions given within a term definition language as referred to in chapter 3. For the BACK system in its present state, handling of redefinition concerning these operations is dealt with under the following principles:

The formalism dictates logical consistency of the set of formulas represented by the net.

- At every stage in the course of building the network there should be a consistent version of the net.
- Every new operation is applied *conjunctively* as an additional formula. Its consistency with the existing set is checked and all inheritance consequences are performed immediately.

We will see how this works in an example. Let us assume the TBox has the following contents: A *person* is a *human* which has 0 to infinitely many *friends* which are *humans*. *Musician* is a *human*.

Athlete is a *human*. *Musician_friend_person* is a *person* with *friends* which are *musicians*. *Athlete_friend_person* is a *person* with *friends* which are *humans*.

Written in the TBox interface language:

```
person = primconcept(specializes(human),
                     restriction(has_friend,human,[0,in]))
musician = primconcept(specializes(human))
athlete = primconcept(specializes(human))
athlete_friend_person =
     primconcept(specializes(person),
                 restriction(has_friend,human,[0,in]))
musician_friend_person =
     defconcept(specializes(person),
        restriction(has_friend,musician,[0,in]))
```

Now, consider the following sequel of incrementally added definitions. First we want to modify our definition by telling that *athlete_friend_person* has a more specific VR for the *has_friend* role *athlete* instead of *human*.

```
athlete_friend_person =
        defconcept(specializes(person),
                   value_restriction(has_friend,athlete))
```

The conjunction of the new and the old concept, in this case athlete, is taken for the final definition.

Then we might want to add a concept C1 with

```
C1 = defconcept(specializes(musician_friend_person))
```

The concept *C1* is introduced and inherits all features of its superconcept. Let us assume we want to establish additionally:

```
C1 = defconcept(specializes(athlete_friend_person))
```

For a new value restriction of the *has friend* role the conjunction of both value restriction in question, *athlete* and

musician is taken. A new concept is generated (which can be renamed later) and serves as the new value restriction for *C1*.

As we notice in the example, redefinition of a single concept always yields the most restricted definition as the new definition. The reason for the restrictive way of dealing with additional specifications follows directly from the interpretation of the network in logic.

A straightforward application of the general principle, as shown in the example gives rise to certain decisions we will discuss below.

Our next example for the type of questions arising in a net management system is inheritance maintenance concerning the local definition of roles at concepts. For an explanation we take a look at the connection between concept and role hierarchy: With regard to roles, BACK underwent a development similar to NIKL in moving towards an independent role hierarchy (which has also been called the *enlightened* view of roles) [Kaczmarek et al 86].

Usually, introduction of a primitive net object means explicit determination of its place within the subsumption hierarchy. For an independent role hierarchy we have to specify a new role by explicitly giving its place in the hierarchy and by an additional specification of its domain and range.

Given two seperate taxonomies of roles and concepts, the connection between both hierarchies can also be described from the concept taxonomy view: A role is attached to the domain concept, which has a value restriction and a number restriction. Following this view, roles can also be introduced implicitly along with a concept operation: If a role restriction is introduced for a concept, and the role has not been introduced so far, the concept the restriction is attached to is taken as the role's domain, and the value restriction concept is taken as the role's range. The number restriction remains as an additional condition for a concept which could be called a local condition (although inheritance is applied globally).

As mentioned earlier, one of the main tasks a management system for a network based on the formalism has to accomplish is the maintenance of inheritance conditions. For role restrictions, both kinds of operations, explicit modification within the role hierarchy and implicit restriction locally at concepts, have to be managed in a uniform way.

In principle, all roles of a concept are strictly inherited by all subconcepts of a concept. For the independent role hierarchy, inheritance is straightforward: Domain and range concept of a differentiating role must be subconcepts of domain and range concept of the coresponding role differentiated. To return to our redefinition problem however, we have to deal with inheritance consequences arising from the dependency of local restriction conditions from the explicit role hierarchy.

Taking our previous example, we add another role *has_good_friend* stating, *Person* is a *human* which has 0 to infinitely many *good_friends* which are *humans*,

```
has_good_friend = primrole(domain_range(person,human))
```

and another concept *C2* as a a *person* which has 0 to 3 *friends* and 3 to 6 *good_friends*.

```
C2 = defconcept(specializes(person),
                restriction(has_friend,human,[0,3]),
                restriction(has_good_friend,human,[3,6]),
```

So far, no relation has been specified between both roles *has_friend* and *has_good_friend*. If the additional specification that *has_friend* is differentiated by *has_good_friend*

```
has_good_friend = primrole(differentiates(has_friend))
```

is entered we apply the conjunction principle local at all concepts. For concept *person*, no new information can be

inferred. For *C2* however, we can get a more specific definition by restricting the corresponding number intervals in the following way: the lower interval bound is propagated upwards and the upper interval bound is propagated downwards. The new definition of *C2* is

C2 is a *person*
C2 has exactly 3 (3 to 3) *friends* and 3 *good_friends*.
(*C2* might be renamed to *three_friend_person*)

The example shows how inheritance mechanisms are applied locally in the wake of a global redefinition. These mechanisms can result in more specific descriptions which constitute additional conditions for concepts in the network.

4.1.1.2 Infimum and Supremum Generation

We will now turn to two additional operations, which comprehend features of several concepts, namely the infimum concept generation and the supremum concept generation.

The infimum concept of two concepts is the defined concept which denotes the intersection of the corresponding sets. An appropriate net operation follows directly from the TBox language in chapter 3; it is specified as:

```
C_infimum = defconcept(specializes(C1),specializes(C2))
```

In our previous example we already made use of this operation in generating the common subconcept of *athletes* and *musicians*.

The supremum concept of two concepts is the defined concept wich denotes a unique superset of the union of the corresponding sets. We introduce the construction of a supremum concept only as an operation performed on the network. No special construct is provided for the supremum concept within the term definition language given in chapter 3 in order to keep a compositional

semantics for the language. Nonetheless, we can consider the construction of a superconcept via the following sequel of operations in the network.

We are looking for the supremum of two concepts, or, to put it differently, for a unique concept that generalizes them. Both concepts must be subsumed by the same root. We first look for all common subsumers. The infimum concept of these is a unique subsumer of the two candidates to be generalized. We can add further restrictions to it by taking all common roles into consideration and assigning appropriate value and number restrictions to it. For number restrictions we take the connected covering of the corresponding intervals. For the value restriction we, again, take the generalization (and apply the construction again).

To illustrate our generalizing operation, let us return to our previous example again and add another concept *two/one_friend_person* with

```
two/one_friend_person =
     defconcept(specializes(person),
                restriction(has_friend,human,[2,2]),
                restriction(has_good_friend,human,[1,1])
```

To obtain the generalization *C4* of *three_friend_person* and *two/one_friend_person* we find that *person* is their most special common subsumer. As additional features we can add restrictions for *C4* finally specifying

C4 is a *person* with 2 to 3 *friends* and
with 1 to 3 *good_friends*.

Summarizing, we can state the following properties: The hierarchy of primitive concepts forms a directed acyclic graph

with regard to subsumption. Every concept net partition subsumed by a root forms a lattice, i.e. by infimum and supremum operation every two concepts have a most special common superconcept and a most general common subconcept.

4.1.1.3 Marking Operations

Let us now review the rest of the operations, namely marking operations like disjointness, root, individual marking. The way these operations are dealt with in this context reflects the distinction given in chapter 3 between term descriptions and term restrictions[35].

Following this distinction, we see two kinds of network operations: Operations like concept specialization or role restriction add new information for net objects or relations which can be characterized as *structural* information. The subsumption test performed by the classifier is based exclusively on this type of information. The extent of these operations is exactly equivalent to the extent of the term definition language.

All remaining operations make additional restrictions on to the net which carry no *structural* information in this sense. However, operations of this type might affect subsumption by imposing conditions which have to be accounted for additionally to the basic classification procedure.

A pair of primitive concepts can be marked explicitly to be disjoint, which means that a common subconcept may not be introduced in the concept hierarchy. Explicit disjointness marking should be distinguished from defined disjointness, i.e. disjointness stated by definitional operations: Two concepts which for example have conflicting number restrictions, cannot have a common subconcept because of the coresponding rules for number restriction inheritance. These concepts are disjoint by

[35] Throughout this section we will take the *axiomatic* view for all considerations concerning the formalism.

virtue of their definitions. In the other case, we might have two primitive concepts which contain no conflicting specifications. If, however, we want to prevent introduction of any common subconcepts, these two concepts can be marked as disjoint explicitly (this is done automatically for all root concepts). We have to restrict explicit disjointness marking to primitives, as otherwise we excluded the introduction of concept definitions which are structurally conceivable.

In our example, both concepts

three_friend_person and *two/one_friend_person*

are *defined* disjoint because of conflicting number restrictions. In contrast for the two primitive concepts

musician and *athlete*,

disjointness would have to be specified additionally by explicit marking.

Primitive concepts can be introduced together with two kinds of additional properties, as roots and as individual concepts. A root concept is a primitive concept. It may not have a superconcept. It is marked as mutaually disjoint from all other root concepts. Root concepts serve as a structuring means for the concept hierarchy. They support concept clustering which is an immediate consequence of most modelling approaches [Hayes 78] for a larger amount of definitions.

An individual concept may not have a subconcept. All concept which have an individual concept as a value restriction must have a number restriction of [1,1].

4.1.1.4 Cycles

Some awkward questions concerning cycles arise. Regarding our incremental way of working with TBox specifications, we often want to refer to an object already defined and use it again - *on*

the left side of the definition - to specify some additional features. While in the TBox language discussed in chapter 3 redefinition was excluded for good reasons, an extended interface language for practical access of the network should cover this possibility, and we have already discussed several aspects thereof.

However, as an unpleasant side effect we open up the possibility of establishing cyclic specifications in the network, and we have to deal with them. The question of what the intention of the user was and whether a modelling idea making use of cycles is sensible shall not be discussed here. Our main concern is to make sure that the net editing system, in particular the classifier does not run into trouble when cycles occur.

What kind of cycles can occur in the network anyway? First consider subsumption cycles. Trivially, they should - for both hierarchies - be excluded; the net management system should make sure we are working with a directed acyclic subsumption graph.

Another case can be seen in the following example: Assuming we restrict the value restrictions of two concepts *C1* and *C2* for a role *is_related* as follows:

```
C1 = defconcept(specializes(C),
                value_restriction(is_related,C2))
C2 = defconcept(specializes(C),
                value_restriction(is_related,C1))
```

Independent of the user's intention, the cyclic definition first does not seem to do any harm within the network. Considering the classification procedure however, it becomes obvious that the subsumption relation between both concepts cannot be determined, the classifier will go astray. We have to exclude either the specification that both concepts are defined or the specification of the value restrictions as given above.

There are cases when cycles concerning primitive concepts are troublesome too: If *C1* and *C2* are given as above but are primitive and we specify *C3* with

```
C3 = defconcept(specializes(C1),specializes(C2))
```

again, by completed inheritance, we obtain a cyclic definition for *C3*:

```
C3 = defconcept(specializes(C1),specializes(C2)
                value_restriction(is_related,C3))
```

which cannot be compared successfully on subsumption by the classifier with another defined concept specified similarly. Consequently, specification of a concept like *C3* is excluded by the network management system.

4.1.1.5 Delete Operations

One final remark concerning network deletion operations.

Delete operations in the system's present state are quite radical; for instance the subsumed subnet is deleted for deletion of a single concept. The present status of these operations should not be seen as a complete set of operations for detailed net debugging with appropriate maintenance mechanisms. So far, it is rather a set of global net operations. A full treatment of the problems involved is beyond the scope of this report.

4.1.2 SOCK, a Net Editor for the TBox

We will now turn to a closer look on how a supporting environment for the net management of a knowledge representation system should look like. The demands on this component arise from the task of working on a larger section of a domain and dealing with a dynamically changing or incrementally growing model. As we have explained above, in most cases the user will

not have a fixed number of well-formulated basic definitions at hand. Instead, he probably has an intuitive idea about a wide range of terminological knowledge. Moreover, the full extent of what should be covered by definitions will change during use of the overall knowledge base.

A tool for the TBox should deal with all tasks for the construction of a taxonomy. For the user interface the environment should support interactive operations for editing the network. The environment should provide a clear way of visualizing and inspecting the net contents[36]. Internally, the environment should automatically support all checking and processing steps required for managing the taxonomy. It has to maintain all definitions and transform them into the basic relations of the representation formalism.

The task of providing an appropriate environment for a representation formalism has been identified as an important part of an overall representation system for quite a while; a recent development is discussed in [Abrett & Burstein 86]. Net management for the BACK system is perfomed by the component *SOCE*, a Structure Oriented Concept Editor. SOCE is a managing component that offers a wide range of functions for entering the terminological knowledge into the system in a practicable way. The entities of the domain terminology can be formalized conceptually and entered via a cooperative user interface with prompting and messaging features. As described above, the representation formalism dictates consistency of terminological knowledge with respect to the inheritance rules. The editor must carry out all necessary checks along with the incremental construction of the network.

In the following, we give an overview of the operations supported by SOCE; for a more detailed introduction into the use of the system environment the reader is referred to the BACK-

[36] The role of interface features for AI systems is illustrated in [Bobrow et al 86]: The case study for the distribution of code shows that 42% are devoted to the user interface.

System User's Guide.

Regarding the overall system architecture concerning the user interfaces (see figure 3) we identify SOCE as the basic component of the knowledge base editing system. It should be mentioned that within this chapter only this basic component is described. The full functionality of the interface operations can much better be appreciated by considering it together with the functionality of the BACK Graphics package. All topics concerning this and other specific interfaces are discussed in section 5.
The SOCE component diagram (figure 4) shows four main subcomponents:

- the net operations component
- the consistency and inheritance component
- the net I/O management
- the classifier

4.1.2.1 Net Operations Component

All operations described in the last section are invoked via the net operations component. In general, the following steps are performed when a SOCE operation is processed:

- All names occuring in a specification are checked for correct reference.
- New links are checked for forming subsumption or definition cycles.
- All checks concerning additional marking restrictions (like root, individual, explicit disjointness) are performed.
- The consistency and inheritance component is invoked for dealing with the structural information of the new specification.
- If the operation produces a consistent extension of the TBox the basic relations are established and filed as BACK core relations.

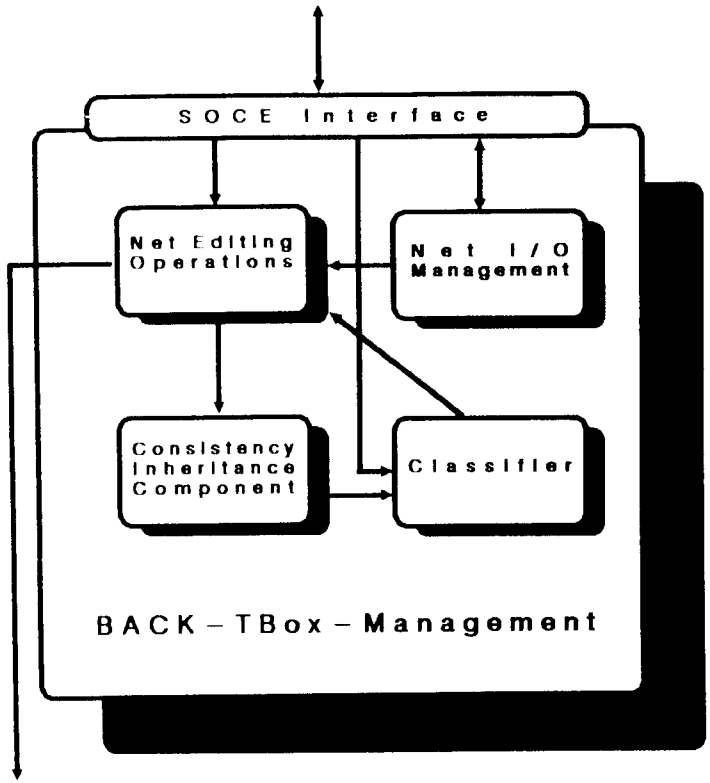


Fig. 4: SOCE Component Diagram

- The SOCE operation is put onto the operations history. This protocol can later on serve as a file for re-loading a net.

Furthermore, let us list some more features for specific operations:

- A number of tests can be performed optionally, which is controlled by system switches. In particular, time consuming tests like disjointness checks can be omitted optionally.

- The editor maintains all user-defined and system-defined names: User-defined concept and role names are kept in a names table, thus providing a *names dictionary* for potential export to other components. System-defined names are created for concepts generated by an infimum operation.
- All user-defined and system-defined names can be renamed; all named objects can be given commentaries as attached data which are not accounted for concerning inheritance.

4.1.2.2 Consistency and Inheritance Component

Within this component the structural consistency of a new specification to the net is checked.

- Conflicts are detected for the concept hierarchy value restriction and number restriction.
- All local modifications concerning role restrictions are checked against the role hierarchy.
- Appropriate mechanisms are applied for inheritance of all roles, diffroles, restrictions, and rvms.
- For multiple inheritance, conflicts concerning value restrictions can make it necessary to generate infimum concepts internally. In this case the classifier is invoked for the system-generated concept, in order to find the right location of the generated concept within the net. The result of classification can also be a merge of the generated concept with a concept already known.
- For the role hierarchy, domain restriction and range restriction conflicts are detected. Domain restriction results in disinheritance of roles at all concepts subsuming the newly specified domain. For domain and range restrictions an internal generation of infimum concepts (i.e. also invocation of the classifier) may be necessary.

4.1.2.3 Net I/O Management

The component encompasses functions for loading, displaying and saving the TBox contents.

The TBox network can be displayed in a pretty-printed mode. The features of all objects are shown according to their specification at concepts and roles. An additional, and for certain purposes much more elegant (and almost *eloquent*) way of visualizing the TBox contents is given by the BACK Graphics package. We will discuss this in 5.2.

All SOCE operations which are performed in the course of a network generation session are recorded and can be saved any time. All operations performed by the classifier, i.e. adding new links and merging of objects, are protocolled as well. The protocol file can also be edited or generated offline.

By loading a protocol file the sequence of SOCE operations is executed. The old TBox contents is added to the new one or is replaced depending on the existence of an initializing operation. The file can also contain TBox interface language expressions. SOCE operations and TBox interface language expressions can be processed in mixed mode. All corresponding checks and inheritance mechanisms are applied in this network loading mode.

Loading and saving a TBox network can also be performed in an alternative way by using the internally filed BACK core relations. The set of core relations is accessible for writing on a file at every stage of the network generation process. Reading a file of core relations again is a quick way to reload a TBox network generated before. No tests are performed at all in this mode.

4.1.2.4 The Classifier

It is the task of the classifier to automatically determine the right place for a term within the taxonomy. Generally speaking, this means that the classifier has to find all subsumption relations between a new description and the network already given.

The core of classification is a comparison between the structural information of two objects; it is most important to see that a proper treatment of this test makes use of the well-defined semantics the formalism is based on. In [Schmolze & Lipkis 83] a classifier algorithm was given; for the BACK formalism the corresponding issues were discussed in chapter 3.

Classification is needed for two main purposes within the system environment: In the modelling phase it is used to establish all subsumption relations between a new specification and the existing network. Later, when querying the TBox, the classification procedure is used to match descriptions in question with apropriate parts of the network structure. These queries are referred to as classification based queries (see 4.1.3.).

A new specification is classified straightforwardly as follows:

- The concept hierarchy is traversed to search for presumptive comparison objects.
- Subsumption is tested.
- For all additionally found relations the appropriate net operations like introduction of new links, and merge of concepts are performed by using the corresponding SOCE operations.
- Since all operations initiated by the classifier are protocolled, a classified net can be stored and re-loaded without using the classifier again.

A classification based query is dealt with in the following way:

- An object is generated following the specification of the queried description.
- The object generated is classified.
- If the object generated is merged with an already existing object the latter is given as result. Otherwise, the generated object is given as answer and remains in the net.
- The other type of queries (net retrieval queries) does not employ the classifier.

Again, considering the different focus of interest we are concerned with under the system view, different questions arise when we investigate classification and its practical use within a knowledge editing environment.

We already know that we want to deal with preliminary specifications or definitions not yet given completely. If classification is designed as a process which is performed automatically for any new TBox specification some unwanted side effects follow: objects which are not completely specified yet could be merged and thus disappear, links could be established which do not correspond to the user's ultimate intention.

Consequently, it should be possible to initiate classification gradually along with the incremental modelling process. The classification process should be fully integrated into the interactive editing environment.

In the BACK system, the classification task is performed combinedly by the SOCE-classifier and the net editor SOCE. SOCE incorporates a number of checking functions to ensure that the new description is placed at an admissible location: the new description must always be consistent with the set of axioms represented by the net. SOCE applies appropriate inheritance mechanisms which further complete the specification and add more restrictions if necessary. In the phase when working exclusively with SOCE the net can be seen as a *pre-classified* net.

In a second step, all additional subsumption relations can be detected by the SOCE-classifier and added to the network.

We will see how division of labor works by reviewing our example: Assume we have the previous TBox contents with

```
has_good_friend = primrole(differentiates(has_friend))

C2 = defconcept(specializes(person),
                restriction(has_friend,person,[0,3],
                restriction(has_good_friend,person,[3,6])
```

and additionally we introduce

```
C5 = defconcept(specializes(person),
                restriction(has_friend,person,[3,3],
                restriction(has_good_friend,person,[3,3])
```

As explained above, SOCE restricts both NRs of *C2* to [3,3] by applying inheritance mechanisms. Both concepts *C2* and *C5* are now identical. Their structural equivalence is not accounted for by SOCE, i.e. both concepts coexist until classification is explicitly initiated. If no other specification has been added the concepts are merged after classification; their names remain accessible as synonyms.

Classification of the overall net is usually not necessary for working with the net. It is possible to remain at the first step, and only let SOCE do all consistency checks incrementally. In this phase, there may exist conceptually identical objects for further specification; the complete set of subsumption relations is not necessarily established at this point. Overall classification can be postponed until the phase in which the full extent of all subsumption relations is needed for full taxonomic reasoning.

Classification can also be performed locally at single concepts: in this case the classifier has to deal with associated VR concepts too, which can result in rearrangement of whole net partitions.

4.1.3 TBox Net Interface Language

Given our TBox language from chapter 3 as a starting point, we will now turn to a formal specification of the complete interface language.

First, we consider two major rationales for the design of an interface language: Every TBox network generated by the language should represent a set of axioms for which a formal interpretation in predicate logic can be given. Consistency should be maintained for the set of axioms represented in this component.

The first part of an interface language for the BACK TBox net was already given in the form of the TBox Language. As shown in chapter 3, both principles hold for the TBox language.

When viewing the set of TBox axioms as a net, we obtain some more operations which are useful for the TBox management. The most important extension is to allow access to object names in order to make redefinition possible. Also operations like supremum generation should be accessible via the TBox Interface Language. Altogether, these operations constitute the set of *Tell* operations for a TBox interface language.

In addition to the specification of Tell operations we will now specify two kinds of *Ask* Operations for the complete design of the TBox Interface Language:

- Net Retrieval Queries
  and
- Classification Based Queries

For net retrieval queries like *what is the value restriction of role R at concept C1*,

```
ASK_VAR = vr(C1,R).
```

the information is directly available from the underlying set of BACK core relations. Classification is not required in this case.

Classification based queries are queries which require a structural matching of a more complex description with parts of the network. For such queries as e.g. *what defined concept specializes C1 and has C2 as its value restriction for role R*,

```
ASK_VAR = defconcept(specializes(C1),vr(R,C2)).
```

the classifier is employed for the matching procedure. A similiar kind of distinction between the process of retrieval and the process of matching is discussed in [Bobrow & Winograd 77].

To summarize, we can give an overall interface specification as an extension of the TBox Language given in chapter 3. All additional operations generate nets (or sets of axioms), which obey the major principles mentioned above.

TBOX-TELL SYNTAX

```
TBoxTellExpr    :== TBoxRestriction |
                    TBoxDefinition

TBoxDefinition  :== Name = TBoxTerm
Name            :== PrologAtom

TBoxTerm        :== Aset | Concept | Role

Aset            :== attrset ( AttributeList ) |
                    attrunion( AsetList ) |
                    Name
AttributeList   :== Attribute |
                    Attribute , AttributeList
Attribute       :== PrologAtom
AsetList        :== Aset | Aset , AsetList

Concept         :== DefConcept |
                    PrimConcept
PrimConcept     :== rootconcept |
                    primconcept( CSpecList ) |
                    Name
DefConcept      :== defconcept( CSpecList ) |
                    Name

CSpecList       :== CSpec | CSpec , CSpecList
CSpec           :== specializes( Concept ) |
                    generalizes( ConceptList ) |
                    value_restriction( Role , ConceptOrAset ) |
                    vr( Role , ConceptOrAset ) |
                    nrmin_restriction( Role , Number ) |
                    min( Role , Number ) |
                    nrmax_restriction( Role , Number ) |
                    max( Role , Number ) |
                    restriction( Role , ConceptOrAset ,
                                 [ Number , Number ] ) |
                    rvm( RvmOp , [ Role ] , [ Role ] )
ConceptList     :== Concept | Concept , ConceptList
RvmOp           :== =
ConceptOrAset   :== Concept | Aset
Number          :== 0 | PositiveInteger | in

Role            :== primrole( RSpecList ) | Name
RSpecList       :== RSpec | RSpec , RSpecList
RSpec           :== differentiates( Role ) |
                    domain_range( Concept ,
                                  ConceptOrAset )
```



```
TboxRestriction :== DisjointnessRestriction |
                    IndividualRestriction
DisjointnessRestriction
                :== disjoint( PrimConceptList )
PrimConceptList :== PrimConcept |
                    PrimConcept , PrimConceptList
IndividualRestriction
                :== individual( PrimConcept )
```

TBOX-ASK SYNTAX

```
TBoxAskExpr     :== PrologVariable = TboxTerm |
PrologVaraible = TboxQueryTerm

TboxQueryTerm   :== node_type_of( Name ) |
                    range_type_of( RoleName ) |
                    sc_of( Name ) |
                    vr_of( ConceptName , RoleName ) |
                    min_of( ConceptName , RoleName ) |
                    max_of( ConceptName , RoleName ) |
                    role_of( ConceptName ) |
                    range_of( Role ) |
                    domain_of( Role )
```

4.2 The ABox Management

The BACK system appears to the user with a textual interface as described in chapter 5. Internally the BACK system ABox management is functionally divided into four components as shown in Fig. 5. These four components are described in detail in the next sections.

4.2.1 The Context Mechanism

The context mechanism of the ABox supports the division of an ABox contents into several, partly dependent contexts. The ideas incorporated are similar to the mechanisms proposed by e.g. Sussman and McDermott in [Sussman & McDermott 72] as part of the programming language CONNIVER[37].

Contexts in the BACK system are partially ordered in context-trees, having the possibility of more than one tree. An assertion valid in one context is valid in all parent contexts. This allows modelling of e.g. possible worlds for planning purposes (cf. e.g. [Luck 85]) or the management of alternatives and was proposed primarily for user-programmable backtrack-mechanisms instead of automatic back-tracking as in PROLOG.

The functionality of such mechanisms is described e.g. in [Barr & Feigenbaum 82], and includes in case of the BACK system the following basic functions:

[37] A simpler version can be found e.g. in FUZZY (cf. [LeFaivre 78]).

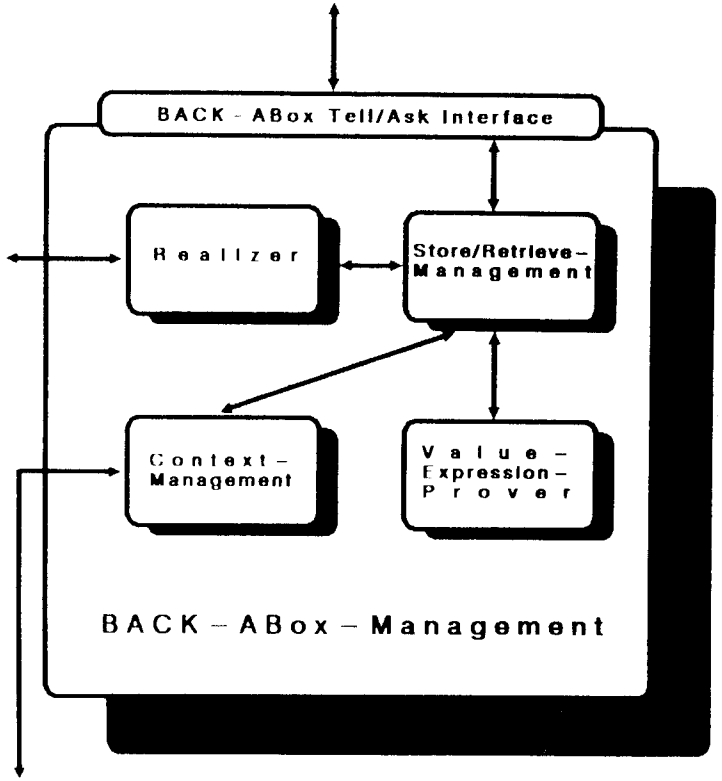


Fig. 5: The ABox Components

- **cntxt_push** :
  Opens a new context which is dependent directly from the actual context and makes this new context the actual one. This context is initially empty, but all assertions accessible in the parent-context are accessible in this new context as well (see the operations clause_all and clause below).

- **cntxt_sprout (<context>)** :
  Opens like push a new context directly dependent from the given context. The new context is not marked as the actual context.

- **cntxt_pop** :
  Pop is the inverse operation to push. The actual context is deleted as well as all contexts depending (directly as well indirectly) from this context. The parent context of the actual context is made the new actual context.

- **switch (<context>)** :
  Makes the context given as an argument the actual context.

- **cntxt_delete (<context>)** :
  Deletes the given context and all (directly as well indirectly) dependent contexts. The actual context remains. if it is not within the set of deleted contexts, otherwise the parent-context of the given context is made the ac'ual context.

- **cntxt_new** :
  Creates a new context which is independend from all other contexts already known within the system. This forces the creation of a new context-tree[38].

For the modification of the contents of one context the following operations are implemented and used within the BACK ABox management system:

---

[38] Such a mechanism of independent contexts was proposed by Brachman and Schmolze in [Brachman & Schmolze 85] as part of the KL-ONE ABox.

- **cntxt_asserta (<context> , <item>)** :
  Asserts the given item in front of all other items in the given context[39].

- **cntxct_assertz (<context> , <item>)** :
  Asserts the given item after all items in this context.

- **cntxt_retract (<context> , <pattern>)** :
  Deletes the first item in the given context fulfilling the specifications of the given pattern[40].

- **cntxt_fetch (<context> , <pattern>)** :
  Unifies the given pattern with the first matchable item of the given context.

- **cntxt_clause_all (<context> , <pattern>)** :
  Unifies the pattern with the first matchable item of the given context. If such a unification fails, this operation is done recursively for all parent contexts of the given context.

- **cntxt_clause (<context> , <pattern>)** :
  Unifies like clause_all the given pattern with the first matchable item of the given context or one of its parents. In contrast to clause_all the behavior of clause in the case of back-tracking differs in that no items are taken as candidates for a next alternative, if a prior item was matchable with respect to the given pattern. This opens the possibility to hide items by 'overwriting' with respect to a given pattern.

---

[39] If no context is given in these operations, the actual context is taken as default

[40] For the pattern-match the PROLOG unification algorithm (cf. e.g. [Clocksin, Mellish 81]) is used.

4.2.2 The Value Expression Prover

The ABox of the BACK system offers a construct called **value expression** for the representation of incomplete knowledge (cf. 3.2.1). In this chapter the management of value expressions is explained in more detail.

The main components of the ABox for handling value expressions are an **expression normalizer** and an **expression prover.** The expression normalizer takes a value expression in a form explained below and produces a normalized value expression corresponding to the definition of value expression given in chapter 3, which describes a disjunctive normal form. The prover takes two value expression and proves a **subsumption relationship** between these two expressions.

So first, the syntax for an unnormalized value expression is given with a semantics oriented to set theoretic semantics as an alternative to the semantics given in chapter 3 for a better understanding of the normalizing process[41].

- A unique constant is a value expression.
- An attribute is a value experssion.
- If VE_1 and VE_2 are value expression, then (VE_1 and VE_2) and (VE_1 or VE_2) are value expression.
- If VE is a value expression, then cwa(VE) and owa(VE) are a value expression.
- If i and j are positive integers (including 0 and infinite) and i is not greater j, then card(i,j) is a value expression[42].

---

[41] In the version of the ABox reported here, disjunctive expressions within cwa are not handled, i.e. the prover fails. This will eventually change in the future (cf. 3.2.1).

[42] The special symbol *in* is provided for representing *infinite*.

- Nothing else is a value expression.

A set-theoretic oriented semantics for value expressions can be given as follows:

If u is a unique constant or an attribute, i,j integer, D the set of all possible unique constants or attributes and P(D) the powerset of D, then AS(VE), the set of alternatives determined by VE, is defined as:

AS(u)	:=	{ m element P(D) ¦ u element m }
AS(card(i,j))	:=	{ m element P(D) ¦ i <= ¦m¦ <= j }
AS(VE_1 and VE_2)	:=	AS(VE_1) intersection AS(VE_2)
AS(VE_1 or VE_2)	:=	AS(VE_1) union AS(VE_2)
AS(owa(VE))	:=	{ m element P(D) ¦ n element AS(VE) and n subset m }
AS(cwa(VE))	:=	AS(VE) intersection P(K(VE))

with K(A) as the set of the unique constants and attributes occurring in the expression VE:

K(u)	:=	{u}
K(VE_1 and VE_2)	:=	K(VE_1) union K(VE_2)
K(VE_1 or VE_2)	:=	K(VE_1) union K(VE_2)
K(cwa(VE))	:=	K(VE)
K(owa(VE))	:=	K(VE)
K(card(i,j))	:=	{}

The normalizing algorithm for value expression is quite straightforward. By inspection of the semantics given, an expression of the form

cwa(x_1 and ... and x_n)

is equivalent to

x_1 and ... and x_n and card(n,n).

The transformation of an expression into its disjunctive normal form is well known and trivial. If an expression has the form

owa(VE),

assuming VE being in a disjunctive normal form, having the form

owa(VE_1 or ... or VE_n),

then this is equivalent to

owa(VE_1) or ... or owa(VE_n).

The form

owa(VE_1 and ... and VE_m)

is equivalent according to the given semantics to

owa(VE_1) and ... and owa(VE_m).

The form owa(u) is equivalent to u, iff u is an unique constant or an attribute. Otherwise, if u equals card(i,j), owa(card(i,j)) is equivalent to card(i,in). If u equals owa(VE), the algorithm is called recursively.

This normalizing algorithm produces a disjunctive normal form according to the syntax for value expression given in chapter 3, if the AS(VE) is not the empty set. If it is empty the value expression is *inconsistent*[43].

The subsumption prover for value expression takes two value expression VE_1 and VE_2 and succeeds stating that VE_1 is **subsumed** by VE_2 ($VE_1 =>_s VE_2$), iff $AS(VE_1)$ is a subset of $AS(VE_2)$. This job is done by comparing the two normalized forms of VE_1 and VE_2, respectively.

For each value expression VE the **cardinality supremum K_{sup}(VE)** is defined as the maximum of the cardinalities of the elements of AS(VE). The **cardinality infimum K_{inf}(VE)** is analogously defined as the minimum of the cardinalities of the elements of AS(VE). From this and the definition of the semantics for value expression it is easy to see that the K_{sup} is monotonic decreasing by conjunction of two value expressions as well as the K_{inf} is monotonic increasing. To put this more formally:

$$K_{sup}(VE_1 \text{ and } VE_2) =< K_{sup}(VE_1)$$
$$K_{sup}(VE_1 \text{ and } VE_2) =< K_{sup}(VE_2)$$
$$K_{inf}(VE_1 \text{ and } VE_2) >= K_{inf}(VE_1)$$
$$K_{inf}(VE_1 \text{ and } VE_2) >= K_{inf}(VE_2)$$

[43] So the normalizing algorithm is a little bit more complicated than sketched here, reducing given value expressions and detecting inconsistent ones.

This result is used within the realizing component (s. [Luck 87]) as a characteristic of the strategy for incremental enlarging an ABox content with the store/retrieve component (cf. 4.2.4). an ABox content with the store/retrieve component (cf. 4.2.4).

4.2.3 The BACK System Realizer

The realizer of the BACK system is responsible for checking items in the ABox for consistency with respect to the definitions made in the TBox. With the same algorithm the taxonomic inferences are drawn on the basis of the TBox definitions.

The main idea of the realizer is the construction of an generic concept definition out of an entry of the ABox. This generic concept definition is incorporated via the TBox-Tell/Ask-Interface (cf. 4.1.3) in the TBox, and will be classified by the classifier (cf. 4.1.2.4).

A generic concept definition is acceptable by the TBox interface with respect to the present content of a given TBox, if it is consistent with respect to that given content. The result of the classifying algorithm, working out the consequences of this generic concept definition, is a **most specific generic (MSG)** concept subsuming the generic concept definition built by the realizer. Therefore the interconnection between TBox and ABox is done only by using the TBox-Tell/Ask-Interface.

The task of the realizer is collecting all relevant information connected to a unique constant for the generation of this generic concept definition[44].

[44] For a discussion of the impact of the structure of an ABox for the realizing process and a comparism of the different approaches within KRYPTON, KL-TWO and BACK for example can be found in [Luck 86].

The details of the BACK system realizer algorithms can be found in [Luck 87].

4.2.4 The ABox Store/Retrieve Component

The ABox store/retrieve-component provides the interface of the ABox to the user. In case of storing new assertions the realizer ensures consistency with respect to the TBox as well as the value expression normaler ensures consistency with respect to value expressions.

In case of retrieving existing assertions the TBox interface is called for subsumption relationships (e.g. searching for a human, an assertion of someone being a man is validation of that) and the subsumption prover for value expression as well.

A formal syntax definition for ABox queries can be given as follows:

```
ABox-Query      :== abox_ask ( VE-Variable - Ask-Function ) |
                    abox_ask ( UniqueConstant - DescrVariable )
Ask-Function    :== a ( Description ) |
                    the ( Description ) |
                    some ( Description ) |
                    all ( Description ) |
                    a ( X , Description(X) ) |
                    the ( X , Description(X) ) |
                    some ( X , Description(X) ) |
                    all ( X , Description(X) )
Description     :== Concept |
                    Concept ( RoleInstances )
Description(X)  :== Concept ( Role = X ) |
                    Concept ( Role = X , RoleInstances )
RoleInstances   :== RoleInstance |
                    RoleInstance , RoleInstances
RoleInstance    :== Role = Value
```



```
Value           :== Value and Value |
                    Value or Value |
                    cwa ( Value ) |
                    owa ( Value ) |
                    card ( Value ) |
                    Val
Val             :== Ask-Function |
                    card ( Min , Max ) |
                    Attribute |
                    UniqueConstant
Min             :== 0 | 1 | 2 | ... | in
Max             :== 0 | 1 | 2 | ... | in
UniqueConstant  :== uc_i
DescrVariable   :== DE_j
VE-Variable     :== VE_k
```

All the ask-functions introduced above refer to the current context of the ABox using the low-level query operator CLAUSE (cf. chapter 4.2.1). Therefore, not the complete contents of the ABox is object of these queries, but only the currently visible part.

In the following, the term **description** is of great importance. A description consists of a TBox concept and a number of TBox roles, each of which are associated with a value expression. In a query, instead of value expressions there can be additional *ask* functions which return value expressions. Furthermore, under certain conditions a free variable may appear instead of a value expression, which may be bound by an *ask function*.

A description is fulfilled by an entry in the ABox, if

- there is a unique constant which is described by a TBox concept that is subsumed by the query description, and
- all the value expressions connected to the unique constant via roles are subsumed by the corresponding value expressions of the query description. A variable in the query description subsumes any corresponding value expression of the unique constant.

This can be expressed more formally as follows:

If D is a description, then C(D) is its corresponding TBox concept, RN(D) are the role names appearing in D, and $RV(RN_i(D))$ is the value expression (or free variable) of $RN_i(D)$.

A description D_{query} is satisfied by an ABox entry, iff
- there is a unique constant described by D_{store},
- $C(D_{query})$ subsumes $C(D_{stored})$,
- for all $RN_i(D_{query})$ there is a $RN_i(D_{stored})$,
- all $RV(RNi(D_{stored})) =>_S RV(RNi(D_{query}))$,

  or
- $RV(RNi(D_{query}))$ is a varibale, which will in this case be replaced by $RV(RN_i(D_{stored}))$.

The ask-functions are one-place functions which are applied to descriptions as arguments (except for the card function, see below). They return a value expression, in case the description is satisfied with respect to the current contents of the ABox. They may be arbitrarily nested, thus allowing for complex queries. The following basic functions are provided:

- **a (<Description>)** :
  Returns the first unique constant that satisfies the description in the current context[45].

- **the (<Description>)** :
  Returns the single unique constant that satisfies the description in the current context. It fails in case there are more than one or none at all.

- **some (<Description>)** :
  Returns all unique constants of the current context satisfying the description in a value expression connected with *or*. This amounts to the union of all alternative sets determined by the unique constants found.

- **all (<Description>)** :

[45] What *first unique constant* means is defined below.

Returns all unique constants of the current context satisfying the description in a value expression connected with *and*. The alternative set of this expression corresponds to the intersection of the alternative sets determined by the unique constants found.

- **a (X, <Description(X)>)**[46] :
  Returns as a value expression the role filler for the free variable X found by projecting the first unique constant satisfying the description in the current context on the Description.

- **the (X, <Description(X)>)** :
  Returns the role filler for the free variable X by projecting the single unique constant satisfying the description on the description. Furthermore, the role filler must be maximally precise and unique. This ask-functions succeeds only if there is a single unique constant or attribute as a role filler for the role with the free variable, or in other words, the value expression for this role filler must be of the form *<Item> and card(1,1)* where Item is either a unique constant or an attribute. If any of these conditions are not satisfied, the function fails.

- **some (X, <Description(X)>)** :
  Returns the value expression resulting from the projection of all unique constants satisfying the description on the description connected with *or*. Again, this corresponds to the union of all the alternative sets of all the value expressions in the position of the free variable. The value expression generated by this function subsumes all the value expressions of the role fillers specified by the free variable of the unique constants matching the description.

- **all (X, <Description(X)>)** :

[46] <Description(X)> is a description with a free variable X, see above.

The same as above, but the value expression returned is the conjunction of all the role fillers found by projection on the description. Thus, the resulting value expression is subsumed by all the value expressions of the role fillers specified by the free variable of the unique constants matching the description.

- **card (<Value Expression>) :**
  Returns a value expression expressing the cardinality of the value expression passed as the argument. The argument may be another Ask-Function.

As already mentioned above, a value expression may appear as a role filler of a role of a description as well as another *ask function* returning a value expression. The evaluation of an Ask-Function is inside-out, so that innermost functions are evaluated first, their results being substituted.

Therefore the ABox-Ask interface can be seen as a function defined as:

ABox-Ask : ABox x Ask-Expression -> Value Expression

Accordingly, the ABox-Tell interface for storing expressions in the ABox can be seen as a function defined as:

ABox-Tell : ABox x Tell-Expression -> ABox

The syntactic structure of a ABox-Tell expression is quite similar to the syntax of the ABox-Ask expressions. The main structure can be given as follows:

```
ABox-Insert    :== abox_tell ( UC-Variable - Description ) |
                   abox_tell ( UniqueConstant - Description ) |
                   abox_tell ( UC-Ref - Description )
UC-Ref         :== the ( Description ) |
                   one ( Description ) |
                   the ( X , Description(X) ) |
                   one ( X , Description(X) )
```



```
Description     :== Concept |
                    Concept ( RoleInstances )
Description(X)  :== Concept ( Role = X ) |
                    Concept ( Role = X , RoleInstances )
RoleInstances   :== RoleInstance | RoleInstance , RoleInstances
RoleInstance    :== Role = Value
Value           :== Value and Value |
                    Value or Value |
                    cwa ( Value ) |
                    owa ( Value ) |
                    card ( Value ) |
                    Val |
                    Ask-Function
Val             :== card ( Min , Max ) |
                    Attribute |
                    UniqueConstant |
                    UC Variable - Description |
                    a ( Description ) |
                    the ( Description ) |
                    some ( Description ) |
                    all ( Description ) |
                    a ( X , Description(X) ) |
                    the ( X , Description(X) ) |
                    some ( X , Description(X) ) |
                    all ( X , Description(X) ) |
                    unique ( Description )
Min             :== 0 | 1 | 2 | ... | in
Max             :== 0 | 1 | 2 | ... | in
UniqueConstant  :== uc_i
UC-Variable     :== UC_i
```

This syntax allows among others the nested creation of new unique constants as e.g.

```
abox_tell(UC1-man(has_friend=UC2 woman))
```

which generates two unique constants, one described as *woman*, the other one described as *man* with, among others has as a friend the before generated unique constant described as *woman*. The ABox content for that assertion looks like

uc_i - man (has_friend = uc_j)
uc_j - woman

The main idea behind the ABox-Tell expression is to connect an unique constant with a description. If the the unique constant

is referenced by a variable, a new unique constant will be created. In all other cases the given unique constant will be described further by the given description.

There are two possibilities to refer to an already known unique constant. First you can name this unique constant by using its internal identifier. On the other hand, you can reference an unique constant by using the functions **the** and **one**. These functions return a unique constant satisfying the given description. The function **the** is as explained before. The semantics for the function **one** can be sketched as follows:

- **one (<Description>)** :
  Returns the first unique constant satisfying the given description and is therefore a synonym to the function **a(<Description>)**.

- **one (X, <Description(X)>)** :
  Returns as a value expression the role filler for the free variable X found by projecting the first unique constant satisfying the description in the current context on the description, which denotes an alternative set with one element which has exactly one element.

The further description of an unique constant can be seen as a logical conjunction of the new description with the already known ones. Therefore the realizing process is monotonic with respect to the specializition of the categories of unique constants.

An insertion in the ABox fails, if the classification process fails. In this case the description of an unique constant is inconsistent with respect to the definitions in the TBox. The other case of failure occurs, if a value expression is inconsistent . In this case the rolefiller of this role was overdetermined.

The semantics of the function **unique** can be sketched as follows:

- **unique (<Description>)** :

  equals the **the(<Description>)**, if an unique constant satisfies the description in the actual context. If this is not the case, a new unique constant is generated described by the given description.

To conclude this presentation of the BACK ABox management system it should be mentioned, that a stack of least recently mentioned unique constants is hold in each context and updated by each insertion of new assertions. This stack is used for interpreting the functions mentioned the *first* occurence of an unique constant. The motivation for this and further details of the BACK ABox can be found in [Luck 87].

5 User Interface

In the development of the BACK system it was an important subtask of the project to establish an environment for the practical usage of the knowledge representation system. In particular, the task of using a specific application domain [Schmiedel et al 86] in order to test several aspects of the formalism was facilitated considerably by the use of additional supporting components which were also developed by the KIT-BACK group.

Besides the usual design considerations for user interfaces in general, the main principle for our conception was to give access to the knowledge base only via a set of operations which guarantee a performance of the system as specified by the semantics of the formalism (see 4.1.1).

In the following we will give an overview of the corresponding user interfaces; for a more detailed introduction the reader is referred to the BACK-System User's Guide. Depending on the hardware environment available the following system components can be used:

- the BACK Basic System[47]
- the BACK Graphics Package[48]

5.1 The Basic System Interface

The basic way of using the system is by accessing the system via the PROLOG interpreter.

[47] the component runs on an IBM 4381 under VM/SP and on SYMBOLICS 36xx

[48] the graphics package runs on SYMBOLICS 36xx

- TBox or ABox are accessed by typing, loading or calling expressions of the TBox or ABox interface language (in the form of PROLOG predicates).
- All functions not supported by these interface languages (e.g. inspecting T/A Box contents) are to be accessed directly as operations of the corresponding component.
- All system output such as T/A Box contents, messages, etc. are displayed line-oriented on screen.
- The SOCE interface is the set of net operations on which interpretation of all TBox descriptions is based. It can be used directly as an additional means of TBox access. In particular an incremental mode of working with the TBox (e.g. during the phase of domain modelling) is supported by this interface. New TBox contents are entered incrementally without automatically classifying the objects which are dealt with. Providing the user with this interface however does by no means stand for free access to the underlying data structure; the SOCE interface can rather be seen as a set of operations specifying the TBox as an abstract data type.
- Using TBox tell expressions instead of SOCE operations is suitable for interfacing with other components or when automatic classification of all newly entered objects is intended.

5.2 The Graphics Interface

In our section describing the representation system, a case was made for having a practically manageable interface. The set of SOCE operations constitute an appropriate interface for the TBox. However, the full functionality can be exploited much better by supporting its usage graphically. Also for the ABox a graphics-oriented utilization of several operations will improve the system's transparancy considerably.

The BACK Graphics package was developed to provide an integrated system envronment for a graphics-oriented usage of the overall system. The package is fully integrated into the Symbolics window management system [Symbolics 85].

- The main choice-menu pane offers the selection of different interfaces according to the mode to be worked in. A choice can be made between entering TBox/Abox interface language expressions, entering TBox SOCE operations graphically, using the natural language test interface for ABox assertions, or using a PROLOG listener for all additional functions.
- The browse window shows the system's contents of interest.
- For editing the TBox a network partition is chosen and displayed as a directed graph. Various partitions can be browsed and inspected stepwise. To achieve a clear visualization of the TBox contents, both concept and role hierarchy are shown apart, one at a time, as pure subsumption hierarchies; a graphical display of all interrelating links can be looked up stepwise.
- Objects of the TBox can be referred to for further specification or for inspection of details by selection with the mouse. Momentary menus are displayed for the choice of further operations; pop-up menus are displayed for the specification of new items.
- For the ABox, the state of the context relations is shown graphically; all context operations can be performed via the browse window.

The figure below shows a view of the graphics interface.

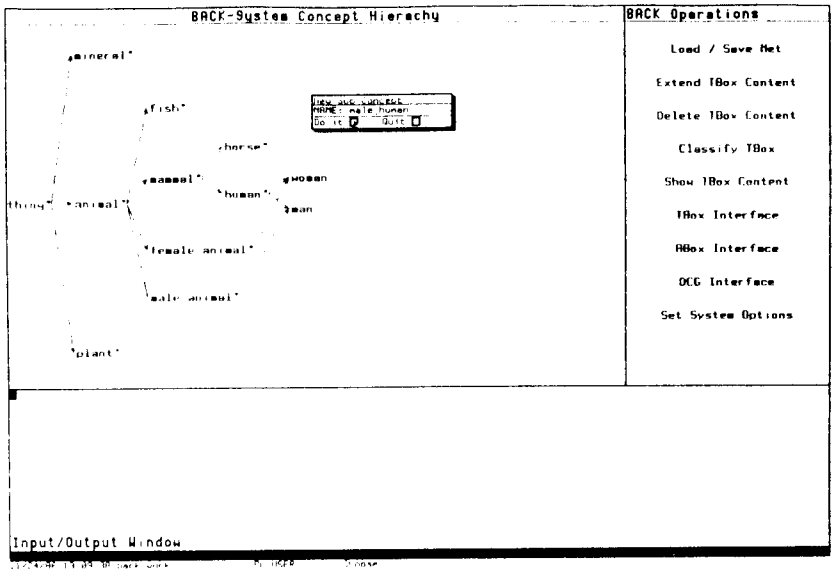


Fig. 6: BACK Graphics Interface

References

[Abrett & Burstein 86]
G. Abrett, M. H. Burstein
The BBN Laboratories Knowledge Acquisition Project: KREME Knowledge Editing Environment
in: DARPA Expert Systems Workshop
Pacific Grove (CA) April 1986 1-21

[McAllester 82]
D.A. McAllester
Reasoning Utility Package User's Manual
MIT AI Memo 667 April 1982

[Anderson & Bower 73]
J. Anderson, G. Bower
Human Associative Memory
Winston 1973

[Barr & Feigenbaum 82]
A. Barr, E.A. Feigenbaum
The Handbook of Artificial Intelligence Vol 2
William Kaufmann 1982

[Bergmann & Gerlach 86]
H. Bergmann, M. Gerlach
QUIRK - Implementierung einer TBox zur Repräsentation von begrifflichem Wissen
Univ. Hamburg, Projekt WISBER, Draft Paper Nov 1986

[Bobrow & Winograd 77]
D.G. Bobrow, T. Winograd
An Overview of KRL, A Knowledge Representation Language
Cognitive Science Vol 1 1977 3-45

[Bobrow et al 86]
D.G. Bobrow, S. Mittal, M.J. Stefik
Expert Systems: Perils Promise
CACM 29(9) 1986 880-894

[Brachman 77]
R.J. Brachman
What's in a Concept: Structural Foundations for Semantic Networks
Int. Journal of Man Machine Studies Vol 9 1977 127-152

[Brachman 79]
R.J. Brachman
On the Epistomological Status of Semantic Networks
Associative Networks
Findler (ed.)
Academic Press 1979 3 50

[Brachman 83]
R.J. Brachman
What IS A Is and Isn't
IEEE Computer 1983 30 36

[Brachman et al 83]
R.J. Brachman, R.E. Fikes, H.J. Levesque
KRYPTON: Integrating Terminology and Assertion
Proc. of AAAI-83 1983 31-35

[Brachman & Levesque 84]
R.J. Brachman, H.J. Levesque
The Tractability of Subsumption in Frame Based Description Languages
Proc. AAAI-84 1984 34-37

[Brachman & Schmolze 85]
R.J. Brachman, J.G. Schmolze
An Overview of the KL-ONE Knowledge Representation System
Cognitive Science Vol 9 1985

[Brachman et al 85]
R.J. Brachman, V. Pigman Gilbert, H.J. Levesque
An Essential Hybrid Reasoning System
Proc. of IJCAI-85 1985 532-539

[Carbonell & Collins 74]
J.R. Carbonell, A.M. Collins
Natural Semantics in AI
Proc. IJCAI-74 1974 344-351

[Clocksin & Mellish 81]
W.F. Clocksin, C.S. Mellish
Programming in Prolog
Springer 1981

[McDermott 76]
D. McDermott
Artificial Intelligence Meets Natural Stupidity
SIGART Newsletter No 57 1976 4-9

[Edelmann & Owsnicki]
J. Edelmann, B. Owsnicki
Data Models in Knowledge Representation Systems: A Case Study
in: Proc. of GWAI-86
Rollinger, Horn (eds.)
Springer 1986

[Emde et al 84]
W. Emde, K. v. Luck, A. Schmiedel
Eine neue Implementation von SRL
Proc. of GWAI-84
Laubsch (ed.)
Springer 1984 219-228

[LeFaivre 78]
R. LeFaivre
FUZZY Reference Manual
Rutgers Univerisity, Comp. Science Dep. 1978

[Freeman et al 83]
M. Freeman, L. Hirschman, D. McKay, M. Palmer
KNET: A Logic-Based Associative Framework for Expert Systems
Burroughs Corp., R/D Div., Technical Report 1983

[MacGregor 86]
B. MacGregor
Personal Communication

[Hayes 77]
P.J. Hayes
In Defence of Logic
Proc. IJCAI-77 1977 559-565

[Hayes 78]
P. Hayes
The Naive Physics Manifesto
in: Expert Systems in the Micro-Electronic Age
Michie (ed.)

[Hendrix 79]
G.G. Hendrix
Encoding Knowledge in Partitioned Networks
in: Associative Networks
Findler (ed.)
Academic Press 1979 51-92

[Kaczmarek et al 86]
T. Kaczmarek, R. Bates, G. Robins
Recent Developments in NIKL
AAAI-86 978-985

[Levesque 82]
H.J. Levesque
A Formal Treatment of Incomplete Knowledge Bases
Univ. of Toronto, Tech. Report CSRG-139 1982

[Levesque 86]
H.J. Levesque
Making Believers out of Computers
Artificial Intelligence Vol 30 1986 81-108

[Lipkis 82]
T. Lipkis
A KL-ONE Classifier,
in: Proc. of the 1981 KL-ONE Workshop br Schmolze, Brachman (eds.)
BBN-Report No 4842 1982 128-145

[Luck 85]
K. v. Luck
Aspekte wissensgestützter Planung
Univ. Hamburg, FB Informatik, Bericht Nr. 110 1985

[Luck et al 85]
K. v. Luck, B. Nebel, C. Peltason, A. Schmiedel
The BACK-System
TU-Berlin, KIT-Report No 29 1985

[Luck 86]
K. v. Luck
Semantic Networks with Number Restricted Roles or Another Story about Clyde
in: Proc. of GWAI-86
Rollinger, Horn (eds.)
Springer 1986

[Luck et al 86]
K. v. Luck, B. Nebel, C. Peltason, A. Schmiedel
BACK to Consistency and Incompleteness
in: Proc. of GWAI-85
Stoyan (ed.)
Springer 1986

[Luck 87]
K. v. Luck
Repräsentation assertionalen Wissens in einem integrierten hybriden System
TU-Berlin, Draft

[Mark 82]
B. Mark
Realization
in: Proc. of the 1981 KL-ONE Workshop Schmolze, Brachman (eds.)
BBN-Report No 4842 June 1982 78-89

[Moore 86]
J. Moore
NIKL Workshop Summary
Boston (MA) USC/ISI Marina del Rey (CA) Draft July 1986

[Moser 83]
M.G. Moser
An Overview of NIKL, the New Implementation of KL-ONE
BBN Annual Report, BBN Rep. No 5421 1983 27-39

[Nebel 86]
B. Nebel
BACK from Europe
Talk given at USC/ISI Marina del Rey July 1986

[Nebel & Sondheimer 86]
B. Nebel, N. Sondheimer
NIGEL Gets to Know Logic: An Experiment in Natural Language Generation Taking a Logical, Knowledge-Based View
in: Proc. of GWAI-86
Rollinger, Horn (eds.)
Springer 1986

[Newell 82]
A. Newell
The Knowledge Level
Artificial Intelligence Vol 18 1982 87-127

[Norman & Rumelhart 75]
D.A. Norman, D.E. Rumelhart
Explorations in Cognition
Freeman 1975

[Patel-Schneider 84]
P.F. Patel-Schneider
Small can be Beautiful in Knowledge Representation
Proc. of the IEEE Workshop on Principles of Knowledge-Based Systems 1984 11-19

[Patel-Schneider 86]
P.F. Patel-Schneider
A Four-Valued Semantics for Frame-Based Description Languages
Proc. AAAI-86 1986 344-348

[Patel-Schneider 87]
P. Patel-Schneider
Decidable, Logic-Based Knowledge Representation
PH.D. Thesis, forthcoming, 1987

[Peltason et al 87]
C. Peltason, K. v. Luck, B. Nebel, A. Schmiedel
The User's Guide to the BACK System
TU-Berlin, KIT-Report No 42 1987

[Quillian 68]
M.R. Quillian
Semantic Memory
in: Semantic Information Processing
Minsky (ed.)
MIT-Press 1968 27-70

[Raphael 68]
B. Raphael
SIR: A Computer Program for Semantic Information Retrieval
in: Semantic Information Processing
Minsky (ed.)

[Robins 86]
G. Robins
The NIKL Manual
USC/ISI, Marina del Rey (CA) April 1986

[Simmons 73]
R.F. Simmons
Semantic Networks: Their Computation and Use for Understanding English Sentences
in: Computer Models of Thought and Language
R.C Schank, K.M Colby(eds.)
Freeman 1973

[Schlumberger 85]
What is a TBox?
Schlumberger Palo Alto Research, The Knowledge Representation Group, Draft 1985

[Simmons 73]
R.F. Simmons
Semantic Networks: Their Computation and Use for Understanding English Sentences
in: Computer Models of Thought and Language
R.C Schank, K.M Colby(eds.)
Freeman 1973

[Schmiedel et al 86]
A. Schmiedel, C. Peltason, B. Nebel, K. v. Luck
'Bitter-Pills' - A Case Study in Knowledge Representation
TU-Berlin, KIT-Report No 39 1986

[Schmolze & Brachman 82]
J.G. Schmolze, R.J. Brachman (eds.)
Proceedings of the 1981 KL-ONE Workshop
BBN-Report No 4842 1982

[Schmolze & Israel 83]
J.G. Schmolze, D. Israel
KL-ONE: Semantics and Classification
BBN Annual Report, BBN Rep. No 5421 1983 27-39

[Schmolze & Lipkis 83]
J.G. Schmolze, T. Lipkis
Classification in the KL-ONE Knowledge Representation System
IJCAI-83 1983 330-332

[Schmolze 85]
J.G. Schmolze
The Language and Semantics of NIKL
BBN Draft 1985

[Schmolze 86]
J.G. Schmolze
Personal Communication

[Stoy 77]
J.E. Stoy
Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory
MIT Press 1977

[Sussman & McDermott 72]
G.J. Sussman, D.V. McDermott
From PLANNER to CONNIVER
Proc. of Fall Joint Comp. Conf. 1972 1171-1179

[Symbolics 85]
Programming the User Interface
Symbolics-Lispmachine Manual No 7 1985

[Tou et al 82]
F. Tou, H.D. Williams, R. Fikes, A. Henderson, T. Malone
RABBIT: An Inteligent Database Assistant
Proc. of AAAI 82 1982 384-318

[Touretzky 86]
D.S. Touretzky
The Mathematics of Inheritance Systems
Morgan Kaufmann 1986

[Vilain 85]
M. Vilain
The Restricted Language Architecture of a Hybrid Representation System
Proc. of IJCAI-85 1985 547-551

[Woods 75]
W.A. Woods
What's in a Link
in: Representation and Understanding
Bobrow, Collins (eds.)
Academic Press 1975 35-82

[Woods et al 76]
W.A. Woods, et al.
Speech Understanding Systems: Final Report
BBN Report No. 3428 1976

[Woods 79]
W.A. Woods
Theoretical Studies in Natural Language Understanding
BBN-Report No 4332 1979